Model-based Monitoring, Diagnosis and Control *

Ph.D. Thesis Proposal

Brown University Department of Computer Science

James Kurien
NASA Ames Research Center
jkurien@arc.nasa.gov

Thesis Committee:

Leslie Kaelbling	**Pandu Nayak**	**Tom Dean**
Brown University	USRA RIACS	Brown University

Abstract

Given a model of a physical process and a sequence of commands and observations received over time, the task of an autonomous controller is to determine the likely states of the process and the actions required to move the process to a desired configuration. We introduce a representation and algorithms for incrementally generating approximate belief states for a restricted but relevant class of partially observable Markov decision processes with very large state spaces. The algorithm incrementally generates, rather than revises, an approximate belief state at any point by abstracting and summarizing segments of the likely trajectories of the process. This enables applications to efficiently maintain a partial belief state when it remains consistent with observations and revisit past assumptions about the process's evolution when the belief state is ruled out. The system presented has been implemented and results on examples from the domain of spacecraft control are presented.

*Funding for this work was provided by the National Aeronautics and Space Administration, through the Cross-Enterprise Technology Development Program.

Contents

Chapter 1

Introduction

This work concerns the automatic control of complex physical systems such as spacecraft or life support systems even in the face of equipment failures or other unexpected events. The first section of this introductory chapter provides a high level introduction to the concept of a control system. The second section motivates why additional research is needed when very successful control systems have been developed for everything from automobile engines to cruise missiles to Furby dolls. The third section briefly introduces the subject of this work, a type of control system that attempts to achieve robustness by performing a significant amount of reasoning about the physical device it controls. The final section introduces the remaining chapters of the document.

1.1 The Impact of Control Systems

Over the past few decades, many common machines have slowly evolved into marvels of functionality, efficiency and reliability. At the same time they have, to the casual observer, kept their familiar forms. Passenger aircraft, looking much like the military transports from which they were derived a half century ago, routinely fly thousands of miles over oceans on autopilot with four, and now only two, engines. Catastrophic failure rates measure once in millions of flights. Automotive disc brakes, first developed in the 1890's, now bring automobiles to a halt on ice or gravel without loss of control and without any special skill on the part of the driver. Internal combustion engines, first used in the mid-nineteenth century, can now power an automobile for a decade without adjustment, with significantly greater power and less pollution than was possible in the recent past. In order to explore one of the factors driving this evolution, let's consider how automobile engines evolved between 1965 and 1995.

An internal combustion engine repeatedly draws air and fuel into an internal chamber where a spark is applied, producing power from the resulting small explosions. While reliable for its day, an engine built in 1965 is a temperamental beast compared to its modern brethren. It's prone to hard starting at extreme temperatures, requires adjustment every few thousand miles, and is apt to spew unburnt fuel and other pollutants from its exhaust. If clogged filters, component failures or deliberate modifications create minor alterations in the way air or fuel are delivered to the engine, it loses power or ceases functioning, leaving the owner to divine what needs to be replaced or adjusted. A 1995 engine starts immediately regardless of conditions, goes 100,000 miles without adjustment, and produces an order of magnitude less pollution than its predecessor. The 1995 engine is impervious to any reasonable change in how fuel and air are delivered. If component failures prevent smooth power production, the engine can often enter a "limp home" mode that reliably produces minimal power, and report the cause of the problem to the user. This revolution in efficiency, reliability and robustness may not seem surprising given how the world can change in three decades. What is interesting to note is that about 80% of the parts that make up the 1965 Ford engine the author has in mind would be indistinguishable to the casual observer from the corresponding parts of the

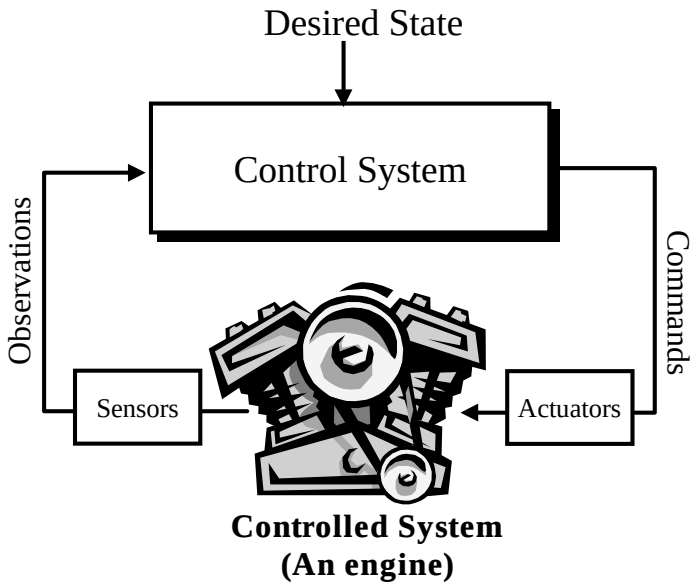


Figure 1.1: Typical Control System Schematic

1995 derivative. Many of the parts are in fact interchangeable. The sole significant difference between these two machines lies in their *control systems*.

As illustrated in Figure 1.1, the purpose of a closed-loop control system is to receive observations from a physical system, estimate the current state of the system, and take actions that move the system to a desired state of operation. This general description of the problem can be applied whether one is attempting to run an engine efficiently, land an airplane automatically, or regulate the human heart with a pacemaker. In the case of the automobile engine, the control system must determine how much fuel to add to the air that is entering the engine and decide at exactly what point in time to apply the spark to keep the process running smoothly. A control system may be extremely simple. In older home thermostats, a metal spring expands or contracts with variation in temperature and an attached switch turns the house's heater on or off. A control system may also be an extremely complex affair, wherein thousands of measurements are used by a team of humans and computers to determine what action to take to control a complex process such as refueling the space shuttle.

Conceptually, the difference between the 1965 and 1995 engines is to what extent the control system of each is aware of the engine's conditions and how flexibly it can respond. The control system for the 1965 engine is designed to virtually eliminate the need for on-board decision making. When the driver depresses the accelerator pedal, a flap opens to allow more air to rush into the engine. The rushing air flows past a fuel source, drawing fuel with it. At the factory, a fuel opening is chosen such that the amount of air that usually rushes through the engine will draw enough fuel to provide adequate performance across a range of usual conditions. If the air flow, air density, or fuel flow change significantly, the opening no longer provides the right amount of fuel. The combustion process loses efficiency or simply stops. The driver then provides the necessary expertise to change the environment to suit the control system, perhaps by letting the car warm up before using it, spraying starting fluid in the engine, or letting excess fuel evaporate.

In contrast, the 1995 engine is controlled by a software system that captures the expertise of Ford's control engineers. Sensors determine the temperature and mass of the air rushing into the engine for each firing. The engine temperature, barometric pressure, and a number of other measurements are also

taken. Based upon these measurements, the control software running on a computer under the dashboard determines how much fuel is required for optimal combustion. It instructs fuel injectors in the engine to open just long enough to spray the desired amount of fuel into the engine, and similarly controls the moment at which the air/fuel mixture will be ignited. In the exhaust stream that results from combustion, oxygen sensors inform the computer whether the ratio of air to fuel being burned is correct. If too little or excess fuel is being delivered, either a sensor or the fuel injector must not be responding properly and commands to the injectors must be adjusted accordingly. Similarly, the computer continually monitors the output of all sensors for indications that a sensor may be malfunctioning. In this case, the output of the sensor is ignored and the computer does its best to operate the engine using only the remaining sensors. Then engine can even be ordered to run through a self test, wherein it changes the commands to the fuel injectors and sparking system and watches for the appropriate responses on the sensors in an attempt to single out problematic components. All failures, whether discovered during normal operations or active testing, are reported to the user and can be downloaded to a diagnostic system along with any anomalous sensor readings for further investigation.

This section was intended to suggest the following intuitions:

- *The job of a control system is to adjust a machine or physical process based upon an estimate of the current conditions of the process, in order to optimize performance.*
- *The estimation method and types of adjustments made in response may be very simple or quite complex.*
- *A machine can be made more robust to failures or environmental change if those changes are identified and the machine is adjusted based upon how the changes will effect its operation.*
- *Increasing the complexity of a machine, by adding sensors, actuators and control software, can paradoxically increase its robustness and the simplicity with which it is operated.*

1.2 Problems in Machine Regulation at NASA

In this work, we will focus on situations wherein the internal state of a machine must be estimated and controlled similar to the engine example. NASA has an endless variety of problems involving the internal regulation of complex machines where the system must continue to operate even in the face of failures. These include operation of human life support systems for Earth-orbit and future missions to the moon and Mars, operation of automated propellant production systems on Mars to enable future exploration, and diagnosis and control of vehicles in the atmosphere, Earth orbit or deep space. Given these exciting and critical applications for control systems, an important question to ask is, why aren't existing methods for developing control systems adequate? The answer lies in the significant differences between these applications and those handled so successfully by industry.

Most importantly, the economics at NASA and a manufacturer such as Ford Motor Company are reversed. Ford released version one of its electronic engine control system in 1984, and by the time version five was released in 1995, it had been installed in tens of millions of relatively similar engines in Ford vehicles. A large amount of effort could be expended to develop the initial control system, as that cost would be amortized over many automobiles. Detailed analysis by engineers to improve performance or reduce per unit cost could potentially be justified by the profit produced by selling millions of units. In addition, new versions of the system could be developed incrementally at a relatively slow rate, drawing upon the experience gained from running millions of copies of the system under varying conditions for several years. At NASA, the current practice is to develop each spacecraft design almost from scratch over a period of two to three years and produce only one or two copies of each design. The fully integrated spacecraft system will often be run for only a few hundred hours or not at all before being deployed in space. Thus our requirements upon the control system development process include:

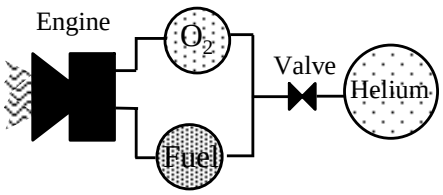


Figure 1.2: Simple propulsion system

- *Control systems must be developed cheaply and quickly in parallel with the hardware system.*
- *They must also be easy to modify once the fully integrated system is tested and deployed.*

A second important distinction is the range of failures over which the control system is expected to operate. If an automobile engine experiences a severe failure or a set of failures that was thought to be highly unlikely, the control system need not continue functioning. The driver can consult the owner's manual for a solution or the automobile can be towed to a shop and repaired. A small amount of down time over the life of an automobile is potentially acceptable, and understandable if a primary component fails. In contrast, many NASA systems such as deep space probes travel far beyond the reach of easy repair. If a component fails very early in a long mission, the control system must continue performing state estimation and control as best it can without that component. In addition, there are critical periods when a short down time will render useless a multi-year spacecraft mission costing hundreds of millions of dollars. For example, if a failure were to prevent a spacecraft from properly decelerating as it approaches a planet or other body it's attempting to orbit, it would burn up in the atmosphere (if any) or be flung uselessly into space. For spacecraft attempting to orbit the more distant planets, by the time mission controllers on Earth received a radio signal indicating that something was amiss, it would be too late to respond. Spacecraft must therefore carry their spare parts with them in the form of multiple copies of critical components (called *block redundancy*) or multiple methods for achieving the same control action using different components (called *functional redundancy*).

Example 1 Consider the schematic of a simple, notional rocket propulsion system shown in Figure 1.2. The purpose of the system is to provide just the right amount of acceleration by combining fuel and an oxidizer in an engine for a specified amount of time. The helium tank is filled with helium under high pressure. Conceptually, the control problem is quite simple. When the valve is opened, the high pressure normally forces oxygen and fuel into the engine where it is ignited to produce thrust. When sufficient thrust is achieved, the valve is closed. While this system is simple, it has the disadvantage that if any component fails, it ceases to operate.

Figure 1.3 illustrates the redundant propulsion system used in the Cassini spacecraft, designed to last a seven year cruise to Saturn and autonomously insert itself into orbit around Saturn. Two engines are provided in the case that one fails. Each engine is supplied with fuel and oxidizer through a complex arrangement of valves. Valves or pipe branches in parallel ensure that if valves stick closed, a redundant parallel valve can be used to allow fluid flow. Valves in series ensure that if valves stick open, an upstream valve can be closed to prevent fluid flow. Not shown are valve drivers that control the latch valves and a set of flow, pressure and acceleration sensors that provide partial observability of the system. There are approximately 10^{15} possible configurations of the system including failures. Several hundred of those configurations produce thrust, depending upon which valves are open or closed. Given a set of failures, thrust configurations that can be reached without using a pyro valve are preferred, as pyro valves can only be opened or closed once. Regardless of the number of failures that occur, we'd like the control system to determine the current configuration of the propulsion system and find the best viable configuration of the

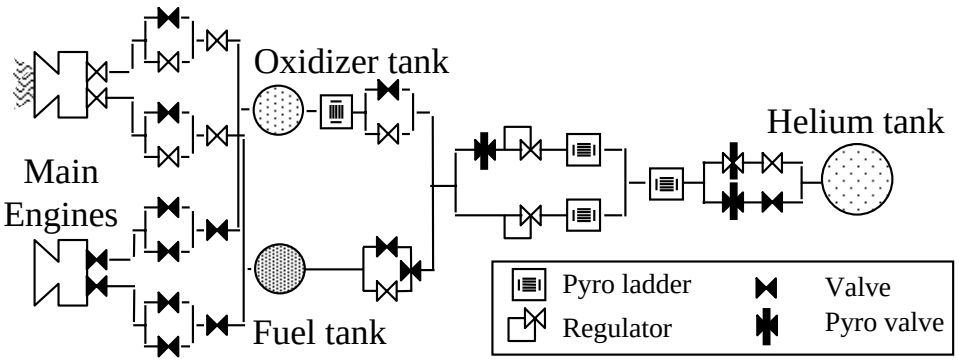


Figure 1.3: Cassini propulsion system schematic.

system that will produce thrust.

The desire for robust operation over long periods without repair, and the resulting complex, redundant systems, introduce additional control system requirements:

- *Due to the number of combinations of failures that could occur over time, the control system must be able to control the system from a very large number of possible states.*
- *Due to the large number of possible states, the control system cannot explicitly record what action to take in each state.*
- *The control system must determine the best action, rather than simply a sufficient action, to take in order to reach the goal configuration.*
- *The control system must include discrete decisions (e.g., Should a valve be opened or closed?, and Should engine A or B be fired?)*
- *Because number of sensors is limited compared to the complexity of the system, the state of the system will not be directly observable from the sensors. The control system will need to generate an estimate of the state and act upon it.*

The first three requirements above suggest that a control system cannot explicitly encode what action to take for each possible combination of sensor readings it receives. Instead, it must have at its disposal a general method for determining the best action to take given the sensor readings. Conceptually, such a general method can be quite simple. For example, if we could directly measure the amount of air flowing into an internal combustion engine, basic chemistry would require that we provide fuel in the ratio of 14.7 to 1 to the air. Our parameter (the amount of air) can vary continuously across the real numbers, and the same control law ($fuel = air/14.7$) informs us how to continuously vary the fuel in response. Unfortunately the fourth and fifth requirements make the application of traditional continuous control laws impractical. A continuous mathematical function that takes as an input the current configuration of the valves in the Cassini propulsion system and computes which valve to open or close first would be quite difficult to derive, encode and understand. A continuous mathematical function that takes as input the current sensor readings and returns the position of each valve in the Cassini system, taking into account sensor failures, redundant information from flow sensors along the same pipe, and so on, would be equally unmanageable. What we require is a method of simply and compactly specifying a discrete controller that applies over all possible states of the system.

Because of the number of states, we cannot specify the discrete control system as a table of sensor values and the discrete decisions that must be made in response. Specifying a control system via rules

that determine what action to take, such as in an expert system or in the "if then else" statements of a program, has the advantage that the rules can be fired regardless of the sensor values that are received. The disadvantage is that it can be quite difficult to determine what states rules or program statements actually cover and how the addition of new rules will affect the behavior of any existing rules. In the case of Cassini, spacecraft engineers performed a large amount of analysis to determine the most likely failures of the system, how to diagnose those failures from the sensor values, and the appropriate response to each. While this provided a highly capable discrete control system, the analysis and software development required came at a cost of over a million dollars per critical segment of the mission (*e.g.*, orbital insertion) and the overall development time was several years. Thus our requirements that control systems be fast and inexpensive to develop is not met.

What makes these techniques difficult and expensive to employ on complex systems is that they are aimed at encoding the discrete decision processes that will be used to perform state identification and control. In essence, encoding the process requires the control system developer to perform the decision process by thinking through how a component failure will effect the behavior of the overall hardware system, how that failure will be diagnosable given the sensors, and what the response should be. If the control system must identify and account for failures in sensors or actuators that will change how the overall hardware system responds, performing the system level reasoning required to create the control system can be quite complex. The more components and subsystems comprising the hardware being controlled, the more complex this system level reasoning grows, and the more expensive and less maintainable the resulting encoding becomes.

One approach to avoiding the cost of encoding a complex process is to encode it only once. Computer graphics are a good analogy. An artist using a computer graphics system does not encode a specialized process for drawing a still life. Instead, the artist describes the local properties of the objects in the scene, such as shape, texture and position. In order to generate a photo-realistic picture of the entire scene, the computer applies standard graphics algorithms to the local descriptions of the objects. When the local properties of an object are changed, the algorithms are re-applied and a new scene is correctly generated. In essence the artist describes *what* is in the scene, while the graphics algorithms capture *how* to draw any scene that can be described in a scene description language. Similarly, our approach will be to introduce standard, reusable algorithms for discrete state identification and control that build upon techniques from the model-based diagnosis literature. To use the algorithms, the control system developer will describe, or model, the local characteristics of the components of a hardware system using a modeling language. These local models will then be combined by the algorithms to perform system level state identification and control over any state the model can attain. We will refer to the resulting system as a model-based discrete control system. The nature of the models and algorithms that make up a model-based discrete control system, and how they satisfy all of the requirements we have introduced, are the subject of the next section.

1.3 Model-based Discrete Control Systems

In order to develop a model-based discrete control system, we require a language for specifying a model of the components of our hardware, and a set of algorithms that make use of our models to perform control. This section provides an overview of the models and algorithms used in this work through an example. *Mention Livingstone*.

Example 2 Figure 1.4 represents a simple valve system that will be used as an example throughout this document. The helium tank pressurizes the system and the valves, if open, allow a gas flow. The valve driver unit (VDU) commands the two valves via the data bus represented by dashed lines. The valves are commanded in parallel. The VDU can command both valves open or both valves closed.

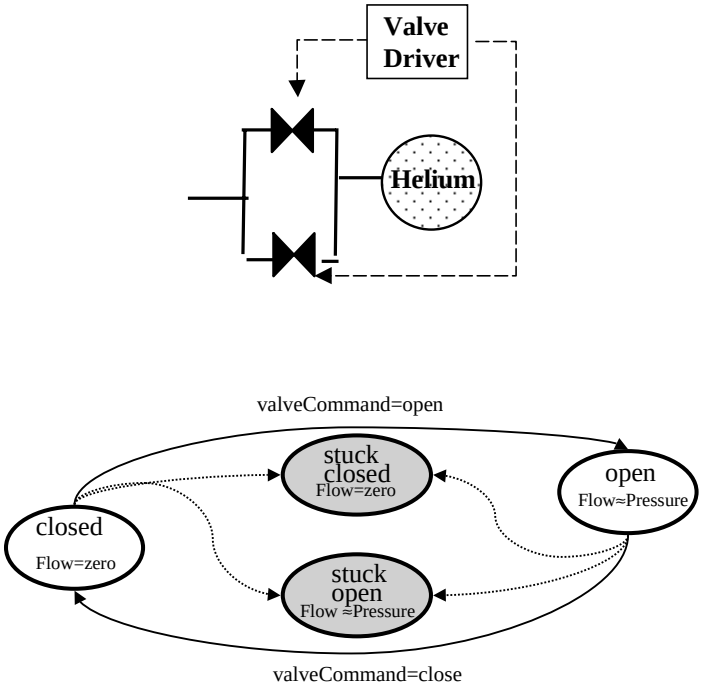


Figure 1.5: The Automaton Representing a Valve

A model of a this hardware system must specify the components of the system (*e.g.,* there are two valves, a tank and a driver). For each component, the model must specify the possible states, referred to as *modes*, the component may occupy (*e.g.,* a valve may be open, closed, stuck open, or stuck closed). For each mode, the model must specify the component's behavior (*e.g.,* a closed valve prevents flow) and transitions (*e.g.,* when commanded open, a closed valve usually opens but may stick closed with some probability p). All of this information can be encoded using an automaton to represent each component. For example, a valve might be represented as shown in Figure 1.5.

The ovals in Figure 1.5 represent the possible modes of the valve, *open*, *closed*, *stuck open* and *stuck closed*. Each mode includes a partial description of how the valve behaves in that mode. For example, when the valve is in the closed mode, the flow through the valve is zero. The arcs specify how the mode changes when an action is taken. Starting in the closed mode, when the command to open is given, the most likely outcome is that the valve moves to the open mode via the darker arc. The lighter arcs represent less likely failure transitions to the stuck open or stuck closed mode that may occur.

Using a single component, we can develop a basic intuition for how a discrete control algorithm might work. For the sake of simplicity, the algorithm is broken down into a method for estimating the state of the system given the sensors and a method for determining the best action to take given the state estimate. Suppose we know a valve to be in the closed mode, and we issue the command to open the valve. We then receive an observation of the flow and pressure, and wish to determine the new state of the valve. Suppose the flow reported by the sensor is zero and the pressure is high. We investigate each possible transition from the closed mode in turn. If the valve took the likely transition to the open mode, the flow through the valve would be proportional to the pressure. It is not, so this transition, although likely, is ruled out. Similarly, the less likely transition to the stuck open mode is ruled out by the observations. The only transition consistent with the observations is the stuck closed mode, and this becomes our state estimate. The basic intuition is that *state identification is a search over the transitions of the hardware model to find a mode that is consistent with the observations*.

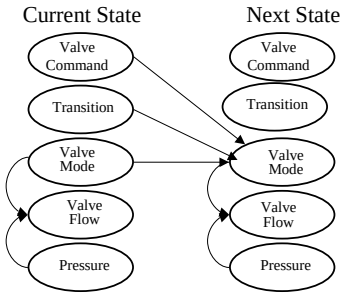


Figure 1.6: An Influence Diagram for the Valve

Choice of a control action is accomplished in a similar manner. Suppose we again know the valve to be in the closed mode. We wish to have flow through the valve. We first check to see if the current mode allows flow. It does not, so we must find a path to a mode that does. We cannot use any of the arcs to failure modes in our path, as we cannot force failures to occur. Instead, we must use only the commandable (darker) arcs. In this case, there is only one arc from the closed mode to the open mode. Fortunately, in the open mode there must be flow and our search ends. The basic intuition is that *action selection is a search over the transitions of the hardware model to find a mode that enforces the desired conditions*.

We can represent the basic features of the automaton of Figure 1.5 in an influence diagram, as shown in Figure 1.6. Each arc represents the existence of one or more constraints between two variables. The straight arcs represent that the current mode of the valve, the valve command and the transition taken determine the next mode of the valve. The curved arc represents that in each state the pressure and mode of the valve determine whether there is flow at the valve. As with the automaton, we can perform state identification and action selection by performing a search over the possible transitions and examining how each influences the observations in the next state. We can also develop an automaton for the VDU and helium tank and represent them as influence diagrams. A combined influence diagram for the VDU system is shown in Figure 1.7. There are variables representing the modes and transitions of the two valves, the VDU and the tank, as well as other variables. As in the valve diagram of Figure 1.6, the pressure in the system influences the flow through the valves. In addition, the mode of the helium tank (*i.e.*, OK or ruptured) determines if there is pressure. Similarly, the mode and command input to the VDU determines the commands sent to the valves. As with the simpler model, we can perform state identification and action selection via searches on the transitions. However, there are now four transition which require choices, leading to 64 possible combinations for this small model. This highlights that these searches are *combinatorial optimization* problems: in state identification, we are interested in the best (in this case, maximum probability) combinations of choices for the transitions that make the next state consistent with the observations. In action selection, we can assign a cost to each commanded transition and find the best (minimal cost) combination of transitions that cause the next state to meet the desired conditions. In Figure 1.8 we see the influence diagram for the VDU system as it is commanded four times. Note how the pressure at valve V1 at time step 2 is dependent upon the valve mode at time step 2. The mode is in turn dependent upon the mode at time step 1, the V1 transition at time step 1, and so on. Thus, computing a state estimate for time step 4 requires a search over all of the transitions in the model. We'll refer to a sequence of transitions as a *trajectory*. If there are 64 transition combinations per time step, and we require a trajectory made up of a sequence of four trajectory combinations, then there are 16,777,216 possible trajectories. While the simple method of choosing trajectories by considering each possible trajectory and what observations it predicts is correct, it clearly cannot be used in practice because of the number of possible trajectories. The next section provides an overview of the research completed to address the state

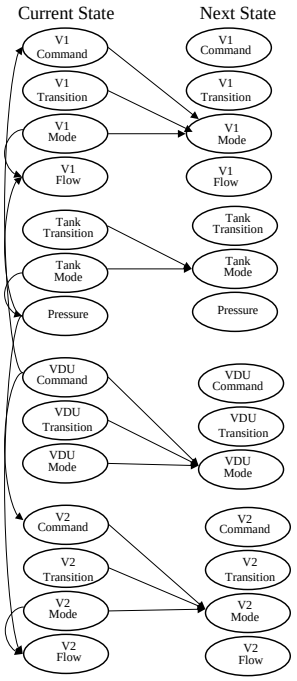


Figure 1.7: An Influence Diagram for the VDU, Two Valves and a Tank. Some arcs not shown for clarity.

identification problem for this model and models with hundreds of components.

1.4 Technical Overview of State Estimation

Given a model of a physical system and a sequence of commands and observations received over time, the task of an autonomous discrete controller is to estimate the likely states of the system and decide the actions required to move the system to a desired configuration. This problem can be formulated as a *partially observable Markov decision process*, or POMDP, a standard problem from operations research and computer science. Given a two-step influence diagram such as Figure 1.7, standard POMDPtechniques can compute a state estimate after an arbitrary number of commands and observations have been received. This state estimate takes the form of a *belief state*, or the likelihood of each possible configuration of the system. The belief state is a sufficient statistic, in that it captures all knowledge about the current state of the system contained within the history of commands and observations. Once the belief state is computed, there is no need to retain or re-examine any previous commands or observations. Unfortunately, computation of the belief state requires computing the probability of every possible configuration of the system, which is out of the question for problems of the size we hope to address.

Rather than computing a complete belief state after each command, we will focus on maintaining the most likely trajectories of the system given the commands and observations received. Since we are only considering a small set of trajectories at any time, it is possible that a new observation will make our set of trajectories unlikely or even inconsistent. The most likely trajectory is then one we have not yet considered. Finding this trajectory will require examination of the history of the system. Thus we will be performing trajectory identification on growing structures similar to Figure 1.7. The naive approach of simply growing the model each time the system is commanded and performing a complete search over the trajectories is clearly insufficient. We will first consider modifying the search algorithm for state identification. Using algorithms inspired by work in the field of model-based diagnosis, we will in the average case vastly speed up the search process. One such technique is to implicitly rule out large numbers of trajectories that are

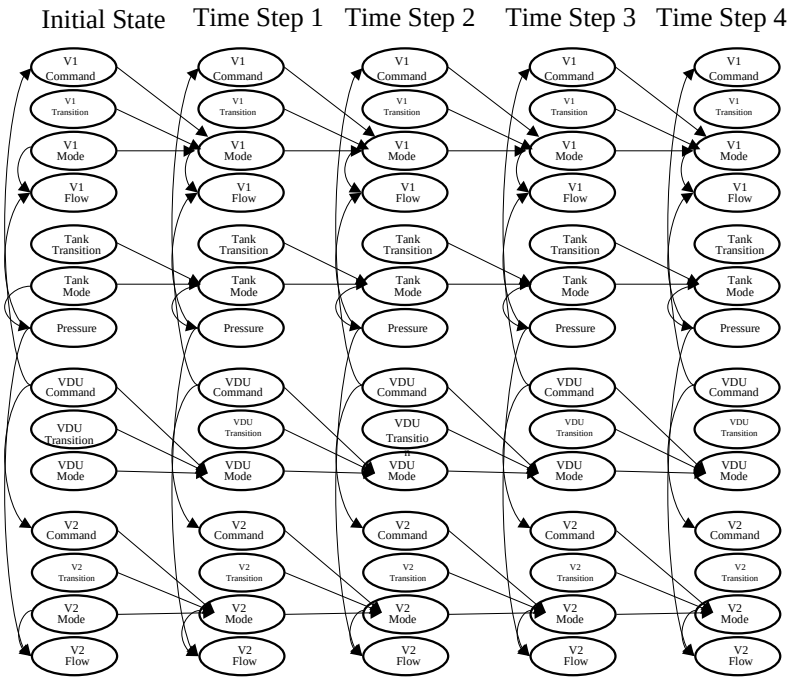


Figure 1.8: An Influence Diagram for the VDU System over 4 Time Steps

inconsistent with the observations without ever explicitly considering them. We will then limit the size of the model that is to be searched through techniques such as not explicitly representing every variable at every time step. These techniques together comprise a practical state estimation algorithm that has been demonstrated on complex models.

1.5 Document Overview

In the following chapters we first address state identification. We begin with a background in state identification from the areas of partially observable Markov decision processes and model-based diagnosis. We then relate state identification to trajectory identification. We introduce the trajectory system representation that implements the intuitions of the influence diagrams presented above in the propositional logic. Then follows a discussion of several search algorithms for the trajectory identification within the transition system representation. Turning to the issue of model size, we introduce optimizations and approximations that prevent the transition system representation from growing unboundedly as time passes. These modifications maintain the ability of a trajectory search algorithm to revise its assessment of how the system evolved in the past in order to reconsider trajectories it had previously dismissed as unlikely. Finally on the subject of state identification we present the results of testing *L2* (for *Livingstone*2), a system that embodies these ideas, on scenarios developed while applying *Livingstone* within NASA and discuss future work. On the subject of action selection, we first present a background from the areas of partially observable Markov decision processes and model-based diagnosis and control. We then outline future work in this area. The final chapter of this document summarizes the results achieved thus far and the work to be completed.

Chapter 2

Related Work in State Identification

In this chapter, we begin discussion of the state estimation portion of a discrete controller. Intuitively, we will be commanding a system that is not completely reliable and will be receiving observations in response. The task of state estimation is to determine the likelihood of each possible state of the system based upon the commands and observations received thus far. Based upon these likelihoods, the appropriate next action may then be selected. We begin the discussion with basic definitions.

Definition 1 A *belief state* is a probability distribution over the possible states of the system. The likelihood assigned to a state by the belief state represents the controller's belief that the system is currently occupying that state. If s is a system state, we will write $b(s)$ for the probability value assigned to state s by belief state b.

Definition 2 The task of state estimation is as follows. Given a model of a system, a sequence of non-deterministic actions taken by the system, and a sequence of observations, compute the belief state.

The state estimation techniques developed in this document draw upon the techniques from two existing formulations of the discrete control problem. The first formulation is the partially observable Markov decision process. The second formulation, model-based diagnosis, is a related but indendently developed formulation of the problem that brings with it a powerful set of algorithms. The two formulations, the basic techniques associated with them, and their shortcomings with respect to our domains of interest are discussed below.

2.1 Partially Observable Markov Decision Processes

A commonly used formalization of the discrete control problem is as a Markov decision process. For a process to be Markov, the current state and action must provide all of the information available for predicting the next state. That is, if we know the current state of the system, knowing the previous state of the system cannot not add information when attempting to predict the next state of the system. We define a Markov decision process as follows.

Definition 3 A *Markov decision process* is defined by the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$, where

- $\mathcal{S}$ is the finite set of states of the system being tracked
- $\mathcal{A}$ is a finite set of actions
- $\mathcal{T}$ is a state transition model of the environment, which is a function mapping elements of $\mathcal{S} \times \mathcal{A}$ into discrete probability distributions over $\mathcal{S}$. The actions are non-deterministic, so we write $T(s, a, s')$ for the probability that the environment will make a transition from state s to state s' when action a is taken.
- $\mathcal{R}$ is a reward function mapping $\mathcal{S}$ to $\Re$ that specifies the instantaneous reward that the agent derives from entering state s. The reward is used in action selection, and is not discussed further in this chapter.

In a Markov decision process, the state of the system is assumed to be directly observable. The probability that an action executed in the current state s will result in a new state s' is determined by $T(s, a, s')$. Once the action is taken, the resulting state is directly observed. Hence there is no state estimation problem. When the state is not completely observable, we must add a model of observations, to create a partially observable Markov model.

Definition 4 A *partially observable Markov decision process* is defined by the tuple $\langle \mathcal{S}, \mathcal{A}, \mathrm{O}, \mathcal{T}, \mathcal{R} \rangle$, where

- $\mathcal{O}$ is a finite set of possible observations
- O is an observation function, mapping $\mathcal{S}$ into discrete probability distributions over $\mathcal{O}$. We write $O(s, o)$ for the probability of making observation o from state s.

Though the current state is not known with certainty in a POMDP, the Markov assumption that knowledge about previous states will not improve our prediction of the next state will still prove useful in designing a state estimator. Such an estimator can be constructed out of T and O by straightforward application of Bayes' rule. Given a belief state b, the output of the state estimator is an updated belief state, b' For each state s', $b'(s')$ can be determined from the previous belief state b, the previous action a, and the current observation o. We will compute $b'(s')$ in two steps. Given our current belief state, we first can compute our new belief the system is in state s' after executing action a, but prior to receiving any observations, denoted $p(s')$,

$$p(s', b) \quad = \quad \sum_{s \in \mathcal{S}} T(s, a, s') b(s) \tag{2.1}$$

This equation is a simple consequence of the Markov property. Intuitively, every state s we could have been in has some likelihood of depositing us into s' given the action a. Each s contributes to s' according to the likelihood we were in state s and the likelihood that that state s transitioned to s'. Once p is computed, we find the new belief in s' conditioned upon the observation o we have received.

$$b'(s') \quad := \frac{O(s', o) p(s', b)}{\Pr(o|a, b)} \tag{2.2}$$

This is simply Bayes' rule. Intuitively, conditionalizing on observation using Bayes' rule redistributes the probability mass according to how much more likely or unlikely it was to see the observed observation o in state s' than in general. $Pr(o \mid a, b)$ is simply a normalizing factor that represents the likelihood of seeing o at all given our previous belief state. Specifically, $Pr(o \mid a, b)$ is the marginalized likelihood of seeing o given action a and our previous belief state b, defined as

$$\Pr(o \mid a, b) \quad = \quad \sum_{s' \in \mathcal{S}} O(a, s', o) \sum_{s \in \mathcal{S}} T(s, a, s') b(s) \quad . \tag{2.3}$$

The resulting $b(s')$ function ensures that the current belief state accurately summarizes all available knowledge. That is, by repeatedly applying Equation 2.2, we maintain a belief state that captures all information contained in an arbitrarily long stream of actions and observations. Thus we have a very simple and elegant solution to discrete state estimation. Unfortunately, it is not practical to directly apply this state estimator in the domains we seek to address.

Consider the problem of determining the likelihood of the possible states of the propulsion subsystem of Figure 2.1. Computing a belief state via Equation 2.2 requires enumeration of the state space. That is, to compute $b(s')$ we must consider $b(s)$ for every $s \in \mathcal{S}$. The propulsion subsystem has 38 components

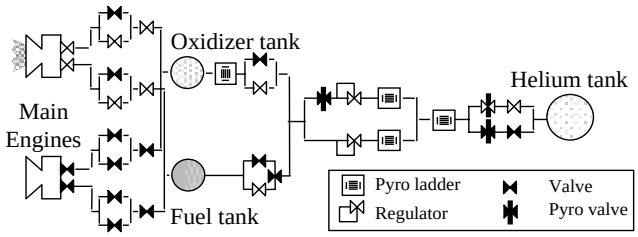


Figure 2.1: Propulsion system schematic.

with an average of 3 states each. The size of $\mathcal{S}$ is approximately 55,000. More complete spacecraft models capture 150 or more components averaging 4 states, yielding a state space of 2^{300} or more [1] and making exact computation of $b(s')$ implausible.

One alternative is to track an approximation whose computation does not require enumeration of the state space. Boyen and Koller (Boyen & Koller 1998), for example, provide an approximate, factored belief state with a bounded error that can be updated without enumerating the state space. Intuitively, the error bound relies upon the stochasticity of the underlying system, parameterized by the problem's *mixing rate*, to continually smear both the approximate and true distributions, exponentially reducing rather than compounding errors over time. Unfortunately, the systems we consider have inadequate mixing rates. Intuitively, when monitoring the internal state of a complex device such as a spacecraft, the device may behave as if it were deterministic for long periods, then exhibit a failure, then return to apparent determinism. There is no process in place with sufficient stochasticity to quickly contract an arbitrary error introduced by a factored approximation.

2.2 Model-Based Diagnosis

Techniques from model-based diagnosis take a different approach, incrementally generating members of the belief state in best-first order. (de Kleer & Williams 1987; 1989). In this approach, the device is typically modeled as a set of components. Each component has a set of variables and one or more states, or modes, that it can occupy. Each mode has a (typically) propositional model that constrains the values of the components variables. Thus, setting the mode of each component induces a set of constraints on the variables of the complete model. Some of these variables are directly observable from the device, meaning that certain assignments of the modes will not be consistent with the observations. The task is then to assign each component's mode so as to cause consistency with the observations.

Component modes that represent failures are assigned a cost corresponding to the prior probability of that failure occurring in that component. An assumption is generally made that failures of components occur independently. Thus the probability of a set of mode assignments is the product of the probability of the mode assignments. Thus starting with the lowest cost assignment (each device in its nominal mode) we can consider all complete mode assignments in order of total likelihood until an assignment consistent with the observations is found. This mode assignment represents the most likely state of the system. Using this simple best-first procedure, many inconsistent mode assignments may be found before a consistent assignment is found. Note however that if a partial assignment to the modes introduces a set of constraints that causes an inconsistency, every full assignment that contains this partial assignment is also inconsistent. This partial assignment to the modes and observations is called a conflict. Recording conflicts as candidate solutions are ruled out and not expanding further portions of the search space that contain them can dramatically focus the search. Conflict-directed best first search, *CBFS*, performs best-

[1] That is, 2,037,035,976,334,486,086,268,445,688,409,378,161,051,468,393,665,936,250,636,140,449,354,381,299,763,336,706,183,397,376

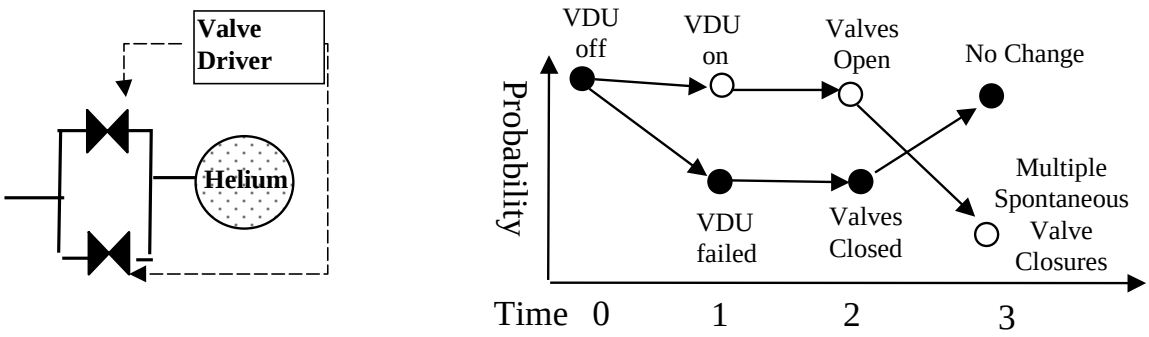


Figure 2.2: Evolution of a Valve Driver Unit and Valves

first search on those parts of the search space not yet known to contain a conflict. Sherlock (de Kleer & Williams 1989) introduces *CBFS* and the application of diagnosing the failure modes within a stateless system such as combinatorial circuits. Conceptually, this type of system tracks for a single step a limited HMM where there is a single action, specified compositionally across the modes, that either has no effect or fails some number of components. The belief state for the HMM after its one action is incrementally generated until the desired number of possible states or amount of probability mass is accumulated.

Livingstone (Williams & Nayak 1996) adds to Sherlock the ability to transition a component's modes between nominal modes in response to an action in addition to moving from a nominal mode to a failure. After each action is performed, *Livingstone* uses *CBFS* to enumerate a small number of most likely mode assignments given the current observations and the previous mode assignments. Each partial belief state is made up of only descendants of the previous n most likely states, which were determined using only previous observations. *Livingstone* tracks the n *approximately* most likely states of the system. This approximation is extremely efficient and well suited to the problem of tracking the internal state of a machine, where the likelihood of the nominal or expected transition dominates, and immediate observations often rule out the nominal trajectory when a failure occurs. The task then becomes one of diagnosing the most likely system transition, chosen from combinations of component transitions, that would be consistent with the unexpected observations. Using this technique, *Livingstone* is able to perform approximate state identification and reconfiguration of systems with hundreds of state variables. It has been applied to the control of a number of systems within NASA and is an integral part of the Remote Agent architecture demonstrated in-flight on the Deep Space 1 spacecraft in 1999 (Muscettola *et al.* 1998; Bernard *et al.* 1998). Unfortunately, the true trajectory may not be among the most likely given only the current observations. Consider the following example.

Example 3 Figure 2.2 reintroduces the valve system from Figure 1.4. Recall that the helium tank pressurizes the system and the VDU commands the valves to open or close in parallel. The graph to the right represents the probability of two possible trajectories. The filled circles represent the true state of the system. At time 0 the VDU is off, the valves are closed and pressure is observed at the outlet of the helium tank. At time 0 the VDU is commanded on. For the sake of illustration, consider an approximate belief state of size 1. The state wherein the VDU is on is placed into the belief state. The true state wherein the VDU is failed is discarded. At time 1, the VDU is commanded to open its valves. Since the only state in the belief state assumes the VDU is on, the single state in the updated belief state has the VDU on and all valves open. In the true, untracked state the valves are closed, as they never received a command. After commanding the valves to open, no flow is observed downstream of the valves. Failure of the helium tank has zero probability, given the observations. Failure of the VDU in the current time step has no effect on the valves. Thus, the most likely next state consistent with the observations requires that all valves sponta-

neously and independently shut. Regardless of the number of valves and the unlikeliness of spontaneous closure, this transition must be taken if it exists. If it does not exist, the belief state approximation becomes empty.

In general, as the true state evolves, the tracked subset of states may need to undergo arbitrarily unlikely transitions in order to remain consistent with the observations. While only one trajectory is tracked in this example, for any fraction of the trajectories that are tracked, an example can be constructed wherein the actual state of the system falls outside the tracked fraction and the error in the approximation may become arbitrarily large.

2.3 Trajectory Identification

In the next chapter, we propose an alternative to committing to a subset of the current belief state or maintaining an approximation of the entire belief state. We propose to maintain the information necessary to begin incrementally generating the current belief state in best-first order at any point in time. Since we do not update the entire belief state, we do not have a sufficient statistic, so a history must be maintained. We introduce a variable to represent every state variable, command and observation at every point in time and an algorithm for incrementally generating the exact belief state at any point. Duplicating the entire set of variables at each point in the history seems impractical except for short duration tasks. We apply two approximations motivated by our experience modeling physical systems for *Livingstone*. The first duplicates only a small number of carefully selected variables at each time point. This approximation is conservative in that does not eliminate any feasible trajectories but may admit certain infeasible trajectories. These may be eliminated by future observations. The second limits the length of the history that is maintained by absorbing older variables into a single variable that grossly approximates them. This allows an approximate belief state to be generated at any point in time from a constant number of variables. The variables represent an exact model of system evolution over the recent past, an approximate model over the intermediate past, and a gross summarization over the more distant past. This allows assignment of the most likely past transitions to be revisited as new observations become available. The fewest variables, and thus the least flexibility, are allocated to segments of the system trajectory that have remained consistent with the system's observed evolution for the longest time.

Chapter 3

Trajectory Identification

3.1 Introduction

We wish to represent the possible histories of a system composed of non-deterministic, concurrent automata given the commands issued to the automata and their output. Figure 3.1 is a version of the valve from Chapter 1, simplified solely for purposes of clarification. From these automata, we would like to create a structure for representing all possible evolutions of the valve over time. We would also like a propositional encoding so we may take advantage of techniques and intuitions developed in the model-based diagnosis world. Based on these desires, in this chapter we introduce a propositional representation of non-deterministic automata that is an extension of the formalism used in *Livingstone*. We then frame the trajectory identification problem.

Before precisely defining the representation, we will develop an intuition using Figure 3.1. Representing the behavior of the automaton within each state is straightforward. Let $Valve$ be a variable representing the possible states of the automaton. The domain of $Valve$ is $\{open, closed, stuck\}$. Let $Flow$ be a variable representing the flow through the valve, of domain $\{zero, nonzero\}$. A propositional model of the open state of the automaton is then simply:

$$Valve = closed \;\; \Rightarrow \;\; Flow = zero$$

The constraints within each of the other states of the automaton can be similarly captured. For our future convenience, we will refer to the set of formulae introduced to model the behavior of each state of each automaton in the system as $\mathcal{M}_\Sigma$. Capturing the transitions of the valve automaton in propositional logic is slightly more challenging. There is no operator to capture that when a command is given to the valve, it non-deterministically chooses with some probability to transition from open to closed or open to stuck closed. A simple way to capture this non-determinism in the valve is to introduce a choice variable τ_{valve}. Figure 3.2 illustrates the augmented automaton. We may now simply model the choice of transitions taken

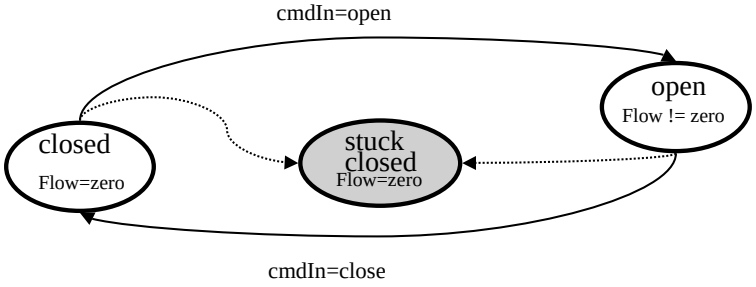


Figure 3.1: A Simplified Valve Automaton

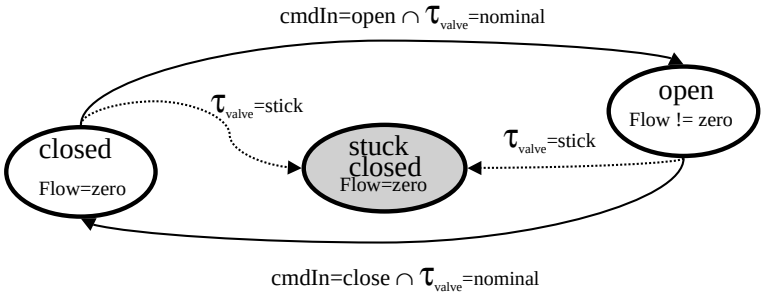


Figure 3.2: Simplified Valve Automaton with τ Variables

from the closed state, for example, as the choice of assignments to the free variable τ_{valve}. To represent the possible outcomes of the open command at time t, we must introduce variables to represent the valve at times t and $t+1$, and a variable to represent the non-deterministic choice τ_{valve} at time t. The transitions from the closed state of the valve automaton can then be modeled by the following formulae:

$$\begin{aligned} Valve_t = closed \wedge \tau_{valve,t} = nominal &\Rightarrow Valve_{t+1} = open \\ \tau_{valve,t} = stick &\Rightarrow Valve_{t+1} = stuck \end{aligned}$$

For our future convenience, we will refer to the set of formulae introduced to model the behavior the transitions of each automaton in the system as $\mathcal{M}_{\mathcal{T}}$. A trajectory of the valve is an assignment to the variables $\tau_{valve,0}, \tau_{valve,1}, \tau_{valve,1}, \ldots$. The prior probability of the valve sticking at any point can be captured as the probability that Nature makes the assignment $\tau_{valve,t} = stick$. Given the appropriate independence assumptions, the set of valve trajectories can be incrementally enumerated in order of prior probability. Trajectories that are inconsistent with $\mathcal{M}_{\Sigma}$ given the observations such as the actual flow observed, need not be considered. This is the kernel of the our approach to state estimation problem. The remainder of this chapter introduces the transition system formalism more precisely and elaborates on the assumptions being made.

3.2 Transition systems

Definition 5 A *transition system* $\mathcal{S}$ is a tuple $\langle \Sigma, \mathcal{T}, \mathcal{D}, \mathcal{C}, \mathcal{M}_{\Sigma}, \mathcal{M}_{\mathcal{T}}, \Gamma \rangle$, where

- Σ is a set of *state variables* representing the state of each automaton. Let *n* denote the number of automata and *m* denote the number of discrete, synchronous time steps over which the state is to be tracked. Σ then contains $m \times n$ variables. Σ_t will denote the set of state variables representing the state of the system at time step t. Each state variable y ranges over a finite domain denoted $\delta(y)$. The temporal variable representing the occurrence of variable y at time step t is denoted y_t.
- $\mathcal{T}$ is a set of *transition variables*. The transition variable that represents the non-determinism in the transition of state variable y from time t to $t+1$ is denoted $\tau_{y,t}$. That is, if there are n non-deterministic outcomes of the transition in the value of y, $\tau_{y,t}$ will have a domain of size n.
- Γ represents a likelihood function on $\mathcal{T}$. The exact nature of Γ is discussed below. Conceptually $\Gamma(\tau_{y,t})$ represents the probability distribution over the outcomes of the transitions of variable y.
- $\mathcal{D}$ is a finite set of *dependent variables*.
- $\mathcal{C}$ is a finite set of *command variables*.
- State s_t is an assignment to $\Sigma_t \cup \mathcal{T}_t \cup \mathcal{D}_t \cup \mathcal{C}_t$
- $\mathcal{M}_{\Sigma}$ is a propositional formula over Σ_t and $\mathcal{D}_t$ that specifies the feasible subset of the state space. A state is feasible if it makes an assignment to $\Sigma_t \cup \mathcal{D}_t$ that is consistent with $\mathcal{M}_{\Sigma}$.

- $\mathcal{M}_{\mathcal{T}}$ is a propositional formula over Σ_t, $\mathcal{D}_t$, $\mathcal{C}_t$, $\mathcal{T}_t$ and Σ_{t+1} that specifies the feasible sequences of states. $\mathcal{M}_{\mathcal{T}}$ is a conjunction of transition formulae modeling possible evolutions of y_t to y_{t+1} of the form

$$\phi_t \wedge (\tau_{y,t} = \tau^*) \Rightarrow y_{t+1} = y^*$$

where ϕ_t is a propositional formula over $\Sigma_t \cup \mathcal{D}_t \cup \mathcal{C}_t$, and τ^*, representing a choice among the non-deterministic transitions of y, is in $\delta(\tau_{y,t})$. The sequence s_i, s_{i+1} is feasible if the assignment made by $s_i \cup s_{i+1}$ is consistent with $\mathcal{M}_{\mathcal{T}}$.

Example 4 We introduce a transition system to model a VDU and two valves. For the sake of brevity we have omitted the helium tank. The variables corresponding to the VDU consist of a state variable vdu representing the possible VDU states (on, *off*, or *failed*), the transition variable τ_{vdu}, a command variable $cmdin$ representing commands to the VDU or its associated valves (on, *off*, $open$, $close$, $none$), and a dependent variable $cmdout$ representing the command the VDU passes on to its valves ($open$, $close$, or $none$). The feasible states of the VDU are specified by the formulae below that belong to $\mathcal{M}_{\Sigma}$.

$$\begin{array}{lcl}
vdu = on & \Rightarrow & (cmdin = open \Rightarrow cmdout = open) \\
 & & \wedge\ (cmdin = close \Rightarrow cmdout = close) \\
 & & \wedge\ ((cmdin \neq open \wedge\ cmdin \neq close) \\
 & & \quad \Rightarrow cmdout = none) \\
vdu = \textit{off} & \Rightarrow & cmdout = none \\
vdu = \textit{failed} & \Rightarrow & cmdout = none
\end{array}$$

together with formulae like $(vdu = on) \vee (vdu = \textit{off}) \vee (vdu = failed)$ and $(vdu \neq on) \vee (vdu \neq \textit{off})$ ... that assert that variables have unique values. The time step subscript is omitted, indicating that all formulae refer to variables within the same time step. $\mathcal{M}_{\mathcal{T}}$ for τ_{vdu} is as follows, where nom is an abbreviation for $nominal$.

$$\begin{array}{l}
\tau_{vdu,t} = nom \Rightarrow \\
\quad \begin{array}{lcl}
vdu_t = off \wedge cmdin_t = on & \Rightarrow & vdu_{t+1} = on \\
vdu_t = off \wedge cmdin \neq on & \Rightarrow & vdu_{t+1} = \textit{off} \\
vdu_t = on \wedge cmdin_t = off & \Rightarrow & vdu_{t+1} = \textit{off} \\
vdu_t = on \wedge cmdin_t \neq off & \Rightarrow & vdu_{t+1} = on \\
vdu_t = failed & \Rightarrow & vdu_{t+1} = failed
\end{array} \\
\tau_{vdu,t} = fail \Rightarrow vdu_{t+1} = failed
\end{array}$$

The valves $v1$ and $v2$ each have a state variable of domain ($open$, $closed$, or $stuck$), a transition variable τ_{vi} and a dependent variable $flow_{vi}$ of domain ($zero$, $nonzero$). The feasible states of the $v1$ are specified by the formula below. The feasible states of $v2$ are specified similarly.

$$\begin{array}{lcl}
v1 = open & \Rightarrow & flow_{v1} = nonzero \\
v1 = closed & \Rightarrow & flow_{v1} = zero \\
v1 = stuck & \Rightarrow & flow_{v1} = zero
\end{array}$$

$\mathcal{M}_{\mathcal{T}}$ for τ_{v1} is shown below. τ_{v2} is as τ_{v1}.

$$\begin{array}{l}
\tau_{v1,t} = nom \Rightarrow \\
\quad \begin{array}{lcl}
v1_t = closed \wedge cmdout_t = open & \Rightarrow & v1_{t+1} = open \\
v1_t = closed \wedge cmdout_t \neq open & \Rightarrow & v1_{t+1} = closed \\
v1_t = open \wedge cmdout_t = closed & \Rightarrow & v1_{t+1} = closed \\
v1_t = open \wedge cmdout_t \neq close & \Rightarrow & v1_{t+1} = open \\
v1_t = stuck & \Rightarrow & v1_{t+1} = stuck
\end{array} \\
\tau_{v1,t} = stick \Rightarrow v1_{,t+1} = stuck
\end{array}$$

3.3 Trajectory Identification

Definition 6 A *trajectory* for $\mathcal{S}$ is a sequence of states $s_0, s_1, \ldots s_m$ such that for all t, $0 < t < m$, s_t is consistent with $\mathcal{M}_\Sigma$ and for all t, $0 < t < (m-1)$, $s_t \cup s_{t+1}$ is consistent with $\mathcal{M}_\mathcal{T}$.

Consider the problem of determining the state of a physical process modeled by a transition system $\mathcal{S}$ at each point in a trajectory $s_0 \ldots s_m$. The subset of the dependent variables $\mathcal{D}$ whose assignment corresponds to a measurement from the process will be referred to as the observations, $\mathcal{O}$. We are given an assignment for the initial state, Σ_0. In addition we are given assignments to commands $\mathcal{C}_t$ and observations $\mathcal{O}_t$ for all $0 < t < m$. The task is to choose assignments to $\tau_{y,t}$ for all y and t so as to ensure consistency with $\mathcal{M}_\Sigma$ and $\mathcal{M}_\mathcal{T}$ and maximize the likelihood of the trajectory. That is to say, given a starting state, a set of commands and a set of observations, we must find the most likely sequence of transitions such that each state is consistent with the state model $\mathcal{M}_\Sigma$ and the transitions are consistent with the transition model $\mathcal{M}_\mathcal{T}$. We define trajectory likelihood to be

$$\sum_{t=0}^{m} \sum_{y=1}^{n} \Gamma(\tau_{y,t})$$

This definition makes the assumption that the likelihood of the assignment to each transition variable is independent of all others. That is, $\tau_{y,t}$ is independent of $\tau_{x,t}$, $\tau_{y,t+i}$ and $\tau_{y,t-i}$. This is a common assumption and has been an adequate approximation in practice. Note that this assumption does not effect the handling of single failures that manifest themselves at multiple points throughout the system (*e.g.,* a power failure causing all lights to go out).

3.4 Infinitesimals

In order to complete the transition system model shown in Example 4, we require the probability of each $\tau_{y,t}$ assignment, representing the prior probability of each possible component transition. Experience with *Livingstone* suggests that an order of magnitude probability scale is sufficient for two reasons. First, the internal behavior of a machine is usually far less stochastic than its interaction with its environment. There is an expected or nominal behavior that a component will exhibit for a given state and input. Failures are one or more orders of magnitude less likely. Second, precise estimates for these priors are often either inaccessible or unknown. In the case of spacecraft, the components may be unique or they may be destined for a new operating environment. However, the relative plausibility of each failure mode during operation can be elicited quite easily. In this work, we formalize and capitalize on these characteristics of the priors by making use of infinitesimals (Goldszmidt & Pearl 1992) to model the relative likelihoods of failures.

An infinitesimal probability is represented by an infinitesimally small constant raised to an exponent referred to as the *rank*. The rank can be considered the degree of unbelievability. Intuitively, one would not consider a rank 2 infinitesimal believable unless all rank 0 and rank 1 possibilities had been eliminated. Composition of infinitesimals has many desirable properties. If A and B are independent events, then

$$Rank(AB) = Rank(A) + Rank(B)$$
$$Rank(A \vee B) = min(Rank(A), Rank(B))$$

Thus an outcome that can occur through multiple independent events has rank i if one event has rank i and the remaining events, even if arbitrarily many, have ranks of i or more. This property is key. It allows us to consider only the most likely trajectories leading to a state: if a sequence of events of rank i ends in state s_j, then an arbitrary number of higher rank (i.e. less likely) trajectories leading to s_j will not change the of s_j. Similarly, if state s_j is reached by a trajectory of rank i, and no trajectory of rank i or less reaches s_k, then s_j is more likely than s_k. We need not consider the possibility that a vast number of unlikely trajectories lead to s_k and together increase its likelihood above that of s_j. Thus $\Gamma(\tau_{y,t} = \tau^*)$ returns the rank of the likelihood of that assignment. We frame our algorithms in terms most likely trajectories, knowing the direct correspondence to most likely states given the infinitesimal interpretation of the priors.

3.5 Correspondence to the POMDPFormulation

The correspondence between a domain specified as a POMDPand a problem P specified as a transition system is straightforward. The state set S of P is the subset of the cross product of the variables of Σ that is consistent with $\mathcal{M}_\Sigma$. Similarly, a set of a system-wide actions $\mathcal{A}$ must be formed from the the factored commands $\mathcal{C}$. If the transition system is limited to receiving one command per time step, then the action set $\mathcal{A}$ is formed by considering each possible value for each command, and augmenting it with the idle value for all other commands. If the model-based controller may issue commands in parallel, then $\mathcal{A}$ consists of the consistent cross-product of the command values. The observation set $\mathcal{O}$ consists of the subset of the cross product of the variables $\mathcal{O}$ that is consistent with $\mathcal{M}_\Sigma$. $\mathcal{M}_\Sigma$ and $\mathcal{M}_\mathcal{T}$ provide a very compact encoding for the observation and transition functions O, T. For a state s and an observation o:

$$O(s, o) = 1 \text{ if } s \wedge \mathcal{M}_\Sigma \models o$$
$$O(s, o) = 0 \text{ if } s \wedge \mathcal{M}_\Sigma \wedge o \models \bot$$

The question arises as to the value of $O(s, o)$ if o is neither inconsistent with nor entailed by $s \wedge \mathcal{M}_\Sigma$. This issue generally arises when using model-based diagnosis algorithms and is not an issue with viewing the problem as a POMDPper se. Often the choice is to make $O(s, o)$ uniform over the consistent values of O. In other cases, algorithms are constructed to implicitly model $O(s, o) = 1$ in the absence of any other information. $O(s, o)$ is then no longer a probability distribution, as it sums to more than one when marginalized on s.

The transition function T is similarly specified. $T(s, a, s')$ is the probability of the assignment to $\mathcal{T}_i$ that transitions the system from s to s', or 0 if no such consistent assignment exists. Given the independence assumptions, the probability of an assignment to $\mathcal{T}_i$ is simply the product of the probabilities of the individual assignments to each variable. physical device. Recall that the belief state update algorithm for a POMDPis derived from the Markov property and Bayes' rule, and is specified by the following formulae:

$$p(s', b) = \sum_{s \in \mathcal{S}} T(s, a, s') b(s)$$
$$b'(s') := \frac{O(s', o) p(s', b)}{\Pr(o \mid a, b)}$$

In the next chapter, we consider a number of belief state generation algorithms for the transition system formulation. These algorithms at their core use the same belief state update algorithm, but take advantage of the structure of $\mathcal{M}_\Sigma$ and $\mathcal{M}_\mathcal{T}$. First, we need not consider any state that is inconsistent with the observations. That is, we may disregard any s' such that $s' \wedge \mathcal{M}_\Sigma \wedge o \models \bot$, as $O(s', o) = 0$ and thus $b'(s') = 0$. If a partial assignment to Σ, c, is such that $c \wedge \mathcal{M}_\Sigma \wedge o \models \bot$, then all states $s' \supset c$ can immediately be eliminated from consideration. The partial assignment $c \cup o$ is referred to as a *conflict* and *conflict-directed* searches use the conflicts they have discovered to greatly reduce the number of states they must examine. Second, $T(s, a, s')$ is the product of the independent probabilities of assignments to members of $\mathcal{T}_i$. Best-first search techniques take advantage of the compositionality of a trajectory's probability to construct assignments to $\mathcal{T}_i$ so as to consider transitions $T(s, a, s')$ in order of probability. Together, these two observations will allow us to construct algorithms that incrementally generate the non-zero members of the belief state in order of probability.

Chapter 4

Trajectory Tracking Algorithms

4.1 The *CBFS-track* Trajectory Tracking Algorithm

The transition-system formulation suggests an intuitive procedure to begin enumerating the belief state at any point. The transition system is initialized with $\mathcal{M}_\Sigma$ and a copy of all variables, representing the initial state. At time step t, we introduce the structure needed to represent the feasible next states of the system. We first create a copy of all variables and extend $\mathcal{M}_\Sigma$, conceptually introducing a new copy of the state constraints where the variables have new time indices, to model the constraints between the variables within the new time step. We then extend $\mathcal{M}_\mathcal{T}$, representing the constraints between the variables in the current and next time steps. Finally, we assign $\mathcal{C}_t$ and $\mathcal{O}_{t+1}$ according to how the system was commanded and the observations that resulted.

Example 5 Figure 4.1 illustrates a trajectory-tracking problem of length three for the model of Example 2.2. Each box represents an variable. The command is $cmdin$ and the observations are $flow_{v1}$ and $flow_{v2}$. These variables are assigned by the problem, as is the start state. The highlighted $\tau_{y,t}$ assignments must be chosen. The remaining variables will be constrained based upon these assignments. The arcs represent constraints from $\mathcal{M}_\mathcal{T}$. Constraints from $\mathcal{M}_\Sigma$ are not shown. For all $\tau_{y,t}$ we will assume $Rank(\tau_{y,t} = nominal) = 0$ and $Rank(\tau_{y,t} \neq nominal) = 1$.

Trajectories may be enumerated in order by enumerating assignments to all $\tau_{y,t}$ in order of the sum of the ranks, then testing for consistency with $\mathcal{M}_\mathcal{T}$ and $\mathcal{M}_\Sigma$. Conflict-directed, best-first search, or *CBFS* (Dressler & Struss 1992; de Kleer & Williams 1989; Williams & Nayak 1996) greatly focuses this process by using conflicts. In this context, a conflict is a partial assignment to $\mathcal{T}$ and $\mathcal{O}$ that is inconsistent. When a candidate solution is found to be inconsistent, the conflict is recorded in a database, ConflictDB. No

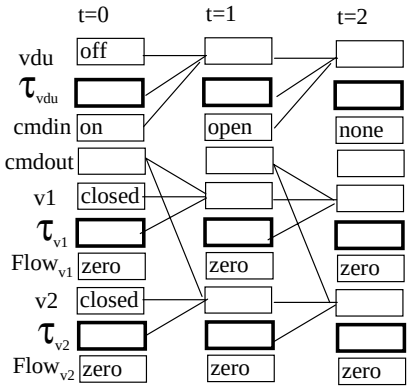


Figure 4.1: Evolution of the VDU/valve system

proc *CBFS-track*()
problem $X = \Sigma_0 \cup \mathcal{D}_0$
Assign Σ_0 to initial state;
loop
$X = X \cup \mathcal{T}_t \cup \mathcal{C}_t \cup \Sigma_{t+1} \cup \mathcal{D}_{t+1}$ to represent new time step $t + 1$;
Assign $\mathcal{C}_t$ and $\mathcal{O}_{t+1}$ according commands and observations received;
Result = n most likely consistent assignments to $\mathcal{T}$ returned by *CBFS* (X, $\mathcal{M}_{\mathcal{T}} \wedge \mathcal{M}_{\Sigma}$, f)
report *Result*;
}
} *CBFS-track*

Figure 4.2: CBFS-based trajectory tracking algorithm

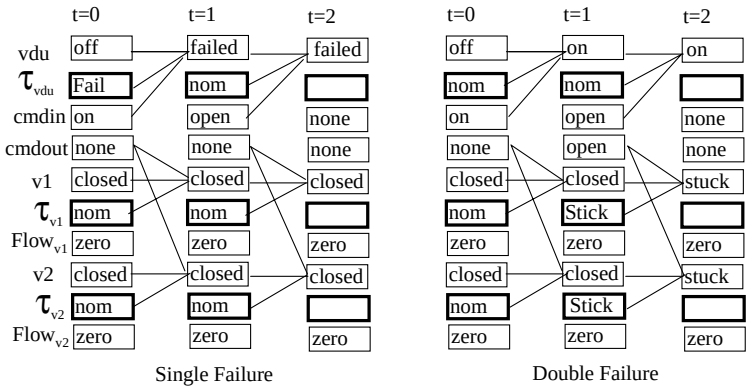


Figure 4.3: Two evolutions of the system

further candidates that contain a known conflict are generated.

We begin with a representation of the initial state of the system in X. At each time step, we extend X with the variables necessary to represent the transition to the next state. We then assign $\mathcal{C}_t$ and $\mathcal{O}_{t+1}$ according to how the system was commanded and the observations that resulted. *CBFS* is then used to enumerate consistent assignments to $\mathcal{T}$ in best-first order. The enumeration could be continued until n assignments are found, until the rank of the next assignment found decreases, or until some other stopping criterion computable from the solutions found thus far is met. The process is then repeated for the next time step.

At each time step, this procedure recomputes from scratch the most likely assignments to the transition variables given all observations. A trajectory t that was most likely given only previous observations might only be consistent with a new observation if it is extended to t' by a very unlikely assignment to the most recently added transition variables. Since *CBFS-track* reconsiders all transition assignments at each time step in best-first order, t' will only be considered if there are no consistent trajectories of likelihood between t and t'. Thus at all time points *CBFS-track* recomputes the most likely members of the belief state given all available commands and observations.

Example 6 Figure 4.3 illustrates the two lowest cost solutions to the above problem would be found by *CBFS*. They represent a single failure of rank 1 at time 1 and a double failure of rank 2 at time 2, respectively.

While this algorithm does compute the most likely states at each time step, it has several significant drawbacks. First, the memoryless quality that allows *CBFS-track* to avoid overcommiting to seemingly likely trajectories also forces the algorithm to rediscover past failures at each time step the system is

proc *Livingstone*(n)
problem $X = \Sigma_0 \cup \mathcal{D}_0 \cup \mathcal{T}_0 \cup \mathcal{C}_0 \cup \Sigma_1 \cup \mathcal{D}_1$
for (i=0;i<n;i=i+1)
$Worlds_i$=initial state;
}
loop
Assign $\mathcal{C}_0$ and $\mathcal{O}_1$ according to new commands and observations received;
for (i=0;i<n;i=i+1)
Σ_0=$Worlds_i$
$\mathcal{T}_0$ = most likely consistent assignment to $\mathcal{T}_0$ returned by *CBFS* (X, $\mathcal{M}_{\mathcal{T}} \wedge \mathcal{M}_{\Sigma}$, f)
$Worlds_i$= Σ_1 as entailed by $\Sigma_0 \wedge \mathcal{C}_0 \wedge \mathcal{T}_0 \wedge \mathcal{M}_{\mathcal{T}}$
}
Report $Worlds$
}
} *Livingstone*

Figure 4.4: Livingstone trajectory tracking algorithm

tracked. Since failures are the exception rather than the rule, we would like a tracking procedure that minimizes computation when no failures have occurred, and when failures do occur, scales the computation required with the inverse of their likelihood. More importantly, the size of the *CBFS* problem to be solved is very large for any given trajectory length and grows unboundedly as the trajectory is extended over time. *CBFS-track* does nothing to elminate variables within a time step based upon the structure of the problem, nor does it attempt to truncate the trajectory representation to maintain a bounded problem size.

4.2 The *Livingstone* Algorithm

The Livingstone system (Williams & Nayak 1996) uses *CBFS* to perform both state identification and control. In this section we will focus on the use of *CBFS* for state identification. While Livingstone was not developed from exactly the transition system formalism described in this paper, it seeks to solve the same underlying problem and can be can be described in this framework.

In order to avoid the problem of an every increasing number of variables to assign that *CBFS-track* encounters, Livingstone does not track the most likely states or trajectories given the commands and observations thus far. It instead approximates the problem of tracking the most likely trajectories given all available information as a recurring trajectory tracking problem of length one. In this problem, the current state is assumed to be known or to be contained in a small set, and the task is to identify the most likely next states given the assignments to $\mathcal{C}_t$ and $\mathcal{O}_{t+1}$. The true current state is then assumed to be in the set of currently most likely states, and the problem recurs. The *Livingstone* algorithm solving this problem is illustrated in Figure 4.4. The parameter n specifies how many states *Livingstone* will track. *Livingstone* in fact represents a class of algorithms that solve the following problem: Given that the system was in one of n states m time steps ago, determine the most likely state given the intervening commands and observations, and then use this as an approximation for the most likely portion of the belief state given the entire trajectory. The current *Livingstone* implementation sets $n = 1$ and $m = 1$ in order to solve problems such as Figure 1.3 with sub-second response time using the relatively weak CPU's found on spacecraft (Bernard *et al.* 1998).

Livingstone does not share the commitmentless property of *CBFS-track* in that it does not reconsider all transition assignments at each time step. It is still the case that a trajectory t that was most likely given only previous observations might only be consistent with a new observation if it is extended to t' by a very unlikely assignment to a transition variable. *Livingstone* considers only the newest transition assignment,

in essense committing to all previous assignments. There is no choice but to extend t to t', even if an assignment of greater likelihood than t' could be found by reconsidering the assignments represented by t. Thus *Livingstone* does not track the most likely states of a system. Rather, *Livingstone* determines the n most likely successors, given the next set of commands and observations, of the states that were the most likely successors given the previous set of commands and observations. Thus each partial belief state is made up of only descendants of the previous approximation of the n most likely states, which were determined using only previous observations. This approach is tractable but fairly vulnerable to ambiguity. Consider the following examples.

Example 7 Recall the VDU system from Figure 2.2. In the first time step we turn the VDU on. From that point onward, the VDU may be on or failed. The only way to distinguish between the two is to attempt to command the valves. Before performing that action, it is much more likely that the VDU is on. *Livingstone* therefore commits to the state wherein the VDU is on. The next state of the system must now evolve from the state wherein the VDU is on. Suppose we now command the valves to open, and receive the observations that there is no flow at either valve. All evolutions from the state wherein the VDU is on and the valves are being commanded open to a state where there is no flow involves all of the valves moving to the stuck state. This holds whether we have two valves or one hundred. Clearly if we had considered the entire trajectory of commanding the VDU then commanding the valves, then conditioning on the multiple flow observations, the most likely trajectory would involve the single failure of the VDU command.

Example 8 Consider a computer that can fail in two ways: its software can hang, in which case it needs to be reset, or its hardware can hang, in which case it needs to be power cycled. Software hangs are significantly more likely than hardware hangs. In either case, the computer fails to respond to keyboard input. Suppose we receive the observation that the computer is not responding to input. It is ambiguous as to which failure the computer is experiencing, so *Livingstone* commits to the software hang. If a reset command fails to revive the computer, *Livingstone* will search for the most likely transition given that the computer was experiencing a software hang, and a reset command failed to revive it. Given the prior probabilities, the most likely failure is a software failure. Since *Livingstone* always considers the most likely extension to the current trajectory, rather than the globally most likely trajectory, there is never an opportunity to consider the hardware failure. *Livingstone* will thus continually consider a software failure, reset, and consider a software failure.

4.3 The Conflict Coverage Algorithm

Note that unlike *CBFS-track* or *Livingstone*, this algorithm should neither rediscover previous failures nor irrevocably commit to a trajectory or set of trajectories that are most likely given the only the current observations. If properly constructed, our procedure will have the following properties:

- It tracks all consistent trajectories at the most likely probability level.
- As long as trajectories at the current probability level remain, very little computation is required.
- As soon as it's no longer consistent to believe the system is in a state at the current probability level, the procedure finds and begins tracking all trajectories at the next probability level.
- Conflicts discovered at each probability level are accrued, ensuring that future conflict-directed searches are highly focused and do not reconsider trajectories that have previously been ruled out.

The strengths of efficiently tracking a partial belief state are merged with the flexibility of incrementally enumerating belief states in the *CoverTrack* procedure of Figure 4.5. $TSet$ is a superset of all consistent trajectories of rank γ, as returned by a previous call to *CoverTrack*. As described above, *extend* adds to the

```
proc CoverTrack(cmd, obs, TSet, ConflictDB, γ) {
    /*Extend the system adding Σ_t to Σ, T_t to T*/
    extend(Σ,T, cmd);
    /*Extend trajectories at current γ */
    Assign T_t to nominal, 0 rank assignment.
    for trajectory in TSet
      trajectory = trajectory ∪T_t;
    /*Check trajectories for consistency, up γ if needed*/
    Assign O according to obs received;
    Survivors = ∅;
    loop{
      for trajectory in TSet {
        conflict=checkConsistency(trajectory);
        if (conflict) then
          push(conflict, ConflictDB);
        else
          push(trajectory,survivors); }
      if(survivors) then return survivors;
      /*Ran out of trajectories. Find more at next rank*/
      γ = γ + 1;
      TSet=GenerateCover(T,ConflictDB,γ);
    }
```

Figure 4.5: Conflict Coverage Tracking Procedure

transition system the variables needed to represent the outcomes of the current command. All trajectories are augmented by the new transition variables, which are assigned nominal transition, and checked for consistency. Any inconsistent trajectory requires additional failures above rank γ, and is discarded as relatively implausible. The survivors are a superset of all consistent trajectories of rank γ. If this set is not empty, it is returned. Otherwise, the most likely trajectory has a rank greater than γ. The *GenerateCover* algorithm generates all assignments to $\mathcal{T}$ of a given rank that cover all known conflicts. A conflict is covered if at least one of the variables in the conflict is assigned to an assignment that does not appear in the conflict. Intuitively, we leave the $\tau_{y,t}$ at their zero rank values, introducing reassignment only to avoid conflicts, with a total cost of γ. This is the NP-hard *hitting set* problem. The contents of $ConflictDB$ and γ will determine whether this problem is tractable. Because of the loss of observations at past time points, *GenerateCover* returns superset of all consistent rank γ trajectories. If at least one trajectory is consistent with the current observations, it is returned. If not, γ is increased.

4.4 Additional Related Work

A more inclusive synthesis of the literature on belief revision and belief update was performed by Friedman and Halpern (Friedman & Halpern 1999). It provides an excellent synthesis of the literature in belief revision and belief update. It describes a general, plausibility-based temporal logic framework that can be used to describe revision methods such described here. The trajectory tracking method described here differs from that described by Friedman and Halpern and the other approximations of which the authors are aware in that it uses history to compensate for not having a sufficient statistic.

Chapter 5

Decreasing the problem size

While applying CBFS or *CBFS-cover* to the full transition system exactly enumerates the most likely trajectories, and thus states, in order, problem size is a significant issue. Let p denote the number of propositions needed to represent each possible value of each variable in $\mathcal{T} \cup \Sigma \cup \mathcal{C} \cup \mathcal{D} \cup \mathcal{O}$. These propositions are constrained by a copy of $\mathcal{M}_{\mathcal{T}}$ and $\mathcal{M}_{\Sigma}$ at each time step. Testing consistency of an m-step candidate trajectory is a consistency problem of $m \times p$ propositions and $m \times \mid \mathcal{M}_{\mathcal{T}} \cup \mathcal{M}_{\Sigma} \mid$ clauses. For the Deep Space 1 model, this is $m \times 4041$ propositions and $m \times 13,503$ clauses.

Let v be the number of ways to choose a variable from $\mathcal{T}$ and assign a failure value ($rank > 0$) value to it. There are $m \times n$ variables in $\mathcal{T}$. Let f denote the average number of failure assignments per variable. Thus, $v = m \times n \times f$ To find the most likely consistent candidate assuming a single failure, the number of consistency checks that would have to be performed on this large theory would be $\mathcal{O}(\mathrm{v})$ in the worst case: any $\tau_{i,t}$ could be assigned to any failure value in its domain. Finding an arbitrary combination of failures would require a number of consistency checks exponential in v.

In this chapter, we reduce the structure needed to represent the evolution of the system at a time point from a complete copy of the system model to a small number of variables and clauses. Intuitively, when a command is issued to the system, only a small number of components participate in transmitting that command through the system or transitioning in response to the command. Consider Figure 5.1. The squares represent state variables, the lines sets of constraints from $\mathcal{M}_{\mathcal{T}}$. As of time 7, the valves, pump and VDU have not been commanded nor have they interacted with other components by passing a command. If we did not detect a failure of any of these components, we can represent the possibility that they remained idle or failed in a localized and unobservable way with a single set of variables and constraints as illustrated. At time 7 we command the valves on. We require variables $v1_8$ and $v2_8$ to represent the new states of the valves. $\mathcal{M}_{\mathcal{T}}$ suggests vdu_7, $v1_7$ and $v2_7$ will interact with $v1_8$ and $v2_8$. These variables, along with necessary transition variables $\tau_{vdu,7}$, $\tau_{v1,7}$ and, $\tau_{v2,7}$, are introduced to the system with the appropriate clauses from $\mathcal{M}_{\mathcal{T}}$. For each other variable y, the variable representing y_7 is adequate to represent y_8. Figure 5.2 illustrates this process. In order to derive a well-founded algorithm from these intuitions, we first place a natural restriction on $\mathcal{M}_{\mathcal{T}}$ that does not impact correctness. Second we introduce an approximation involving $\mathcal{M}_{\Sigma}$ that, importantly, does not rule out consistent trajectories. Instead, some trajectories that are not consistent with past observations may be admitted, with the possibility that future observations will eliminate them. These problem modifications avoid replication of many variables in Σ and $\mathcal{D}$, as well as corresponding constraints from $\mathcal{M}_{\mathcal{T}}$ and $\mathcal{M}_{\Sigma}$.

5.1 Restricting $\mathcal{M}_{\mathcal{T}}$

We restrict $\mathcal{M}_{\mathcal{T}}$ as do *Livingstone* and *Burton* (Williams & Nayak 1997): a component moves to a failure state with equal probability from any state, and except for failures a component that does not receive a

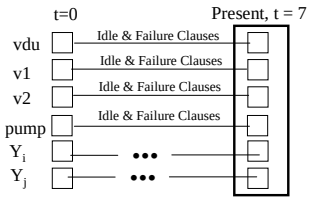


Figure 5.1: Evolution before commanding the valves

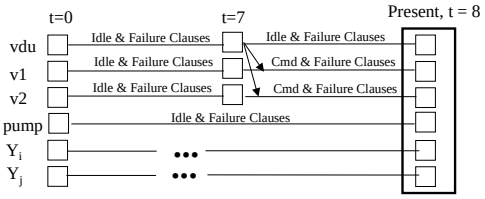


Figure 5.2: Evolution upon commanding the valves

command idles in its current state. $\mathcal{M}_{\mathcal{T}}$ is limited to the forms:

$$\begin{aligned} (\tau_{y,t} = \tau_{fail}) &\Rightarrow y_{t+1} = y_{fail} \\ (\mathcal{C}_{y,t} = \mathcal{C}^*) \wedge \phi_t \wedge (\tau_{y,t} = nom) &\Rightarrow y_{t+1} = y^* \\ (\mathcal{C}_{y,t} = idle) \wedge (\tau_{y,t} = nom) &\Rightarrow y_{t+1} = y_t \end{aligned}$$

where ϕ_t is a propositional formula over $\Sigma_t \cup \mathcal{D}_t$, $\mathcal{C}^* \in \delta(\mathcal{C}_{y,t})$, $nom \in \delta(\tau_{y,t})$ and $\tau_{fail} \in \delta(\tau_{y,t})$. Formulae of the first form model failures while formulae of the second form model nominal, commanded transitions. Formula of the third form are frame axioms that encode our assumption that devices that do not receive a command remain in their current state. We replace ϕ_t with implicant π_t, an equivalent formula involving only Σ_t. Intuitively ϕ_t is a formula involving $\mathcal{D}$ that, given $\mathcal{M}_\Sigma$ and an assignment to Σ, allows us to infer if $\mathcal{C}_{y,t}$ propagates through a set of components to component y. To form π_t, we replace each assignment to $\mathcal{D}_t$ with a set of assignments from Σ_t that imply the $\mathcal{D}_t$ assignment under $\mathcal{M}_\Sigma$. We expect that for the type of clauses $\mathcal{M}_{\mathcal{T}}$ contains, growth in π_t will be proportional to the length of the component chain that transmits $\mathcal{C}_{y,t}$, which ranged from 1 to 5 in (Bernard *et al.* 1998). Our experience supports this hypothesis. This growth is offset as non-idle, non-failure clauses take the following form which is independent of $\mathcal{D}$.

$$(\mathcal{C}_{y,t} = \mathcal{C}^*) \wedge \pi_t \wedge (\tau_{y,t} = nom) \Rightarrow y_{t+1} = y^*$$

Given a $\mathcal{C}_{y,t}$ which is not idle, in order to determine consistency with $\mathcal{M}_{\mathcal{T}}$ we now need only introduce $\mathcal{C}_{y,t}$, $\tau_{y,t}$ and those select members of Σ_t that appear in π_t.

5.2 Eliminating intermediate observations

$\mathcal{M}_\Sigma$ remains, and requires introduction of all variables in Σ_t and $\mathcal{D}_t$ in order to check consistency against $\mathcal{O}_t$. We proceed by eliminating all variables $\mathcal{O}_t$ for values of t sufficiently far in the past. That is to say, transition choices are only constrained by consistency between the trajectories they imply and recent observations. As the system evolves, variables representing older observations and the copies of $\mathcal{M}_\Sigma$ that constrain them are unneeded. For the portions of the trajectory where $\mathcal{M}_\Sigma$ is not introduced, we need not introduce $\mathcal{D}$ and need only introduce the limited portion of Σ_t required by $\mathcal{M}_{\mathcal{T}}$. This is of course an approximation. It is now possible to choose transition assignments that are inconsistent with the discarded observations, resulting in an "imposter" trajectory. This approximation has several important features. First, it is a conservative approximation in that no consistent trajectories are eliminated. Second,

all trajectories are checked against new observations, and impostors are eliminated as soon as they fail to describe the on-going evolution of the system. Finally if conflicts are recorded in $ConflictDB$, no partial assignment to $\mathcal{T}$ that was discovered to be in conflict with the observations will be reconsidered, even after observations are discarded. Thus we can only admit an imposter in the case where a transition choice is in conflict with an observation, but the choice is not considered until after the conflicting observation has been discarded.

5.3 Selective Model Extension

Based upon these restrictions, the procedure *extend* introduces into time step t only the small fraction of the model involved with the evolution of the system due to the command $\mathcal{C}_{y,t} = \mathcal{C}^*$. The resulting problem size per time step is proportional to $\mid \pi_t \mid$. This hinges upon Theorem 1. For the purpose of discussion we will assume that for each time step t there exists only one y for which $\mathcal{C}_{y,t} \neq idle$. The proofs can be extended to parallel commanding.

Theorem 1 *Assume $\mathcal{C}_{y,t} = \mathcal{C}^*$, $\mathcal{C}^* \neq idle$, and for all $x \neq y$, $\mathcal{C}_{x,t} = idle$. Consider the formula of $\mathcal{M}_\mathcal{T}$*

$$(\mathcal{C}_{y,t} = \mathcal{C}^*) \wedge \pi_t \wedge (\tau_{y,t} = nom) \;\; \Rightarrow \;\; y_{y+1} = y^*$$

For all state variables x_t, $x \neq y$, if $x_t \notin \pi_t$, then an equivalent consistency problem is formed by replacing x_t, $\tau_{x,t}$ and all formulae of $\mathcal{M}_\mathcal{T}$ involving these variables with a constraint between x_{t-1} and x_{t+1}.

Intuitively, there are no witnesses to the value of x_t except for x_{t-1} and x_{t+1}, which can be constrained directly. If x_t is as described, then the only clauses involving x_t are of the form:

$$\begin{aligned}
(\mathcal{C}_{x,t-1} = \mathcal{C}^*) \wedge \phi_{t-1} \wedge (\tau_{x,t-1} = nom) &\;\; \Rightarrow \;\; x_t = x^* \\
(\mathcal{C}_{x,t-1} = idle) \wedge (\tau_{x,t-1} = nom) &\;\; \Rightarrow \;\; x_t = x_{t-1} \\
(\tau_{x,t-1} = \tau_{fail}) &\;\; \Rightarrow \;\; x_t = x_{fail} \\
(\mathcal{C}_{x,t} = idle) \wedge (\tau_{x,t} = nom) &\;\; \Rightarrow \;\; x_{t+1} = x_t \\
(\tau_{x,t} = \tau_{fail}) &\;\; \Rightarrow \;\; x_{t+1} = x_{fail}
\end{aligned}$$

The variable x_t can only impact the consistency of the system via the assignments to $\tau_{x,t-1}$ and $\tau_{x,t}$. Given the independence assumptions, assigning failures to both is indistinguishable from and less likely than assigning $\tau_{x,t-1} = nom$ and $\tau_{x,t}$ to a failure, while assigning a failure to one is equivalent to assigning a failure to the other. Thus we need only consider $\tau_{x,t-1} = \tau_{x,t} = nom$ and $\tau_{x,t-1} = nom$, $\tau_{x,t} = \tau_{fail}$. In the nominal case, x_t is equivalent to x_{t+1} and can be eliminated. In the failure case, the assignment to x_t has no impact on x_{t+1} and can be eliminated. The above formula are rendered equivalent to the following reduced set:

$$\begin{aligned}
(\mathcal{C}_{x,t-1} = \mathcal{C}^*) \wedge \pi_{t-1} \wedge (\tau_{x,t-1} = nom) &\;\; \Rightarrow \;\; x_{t+1} = x^* \\
(\mathcal{C}_{x,t-1} = idle) \wedge (\tau_{x,t-1} = nom) &\;\; \Rightarrow \;\; x_{t+1} = x_{t-1} \\
(\tau_{x,t-1} = \tau_{fail}) &\;\; \Rightarrow \;\; x_{t+1} = x_{fail}
\end{aligned}$$

In fact, at time t we will know whether or not $\mathcal{C}_{x,t-1} = idle$, and therefore we need only introduce one of the first two formulae. The $extend$ procedure repeatedly applies Theorem 1 to avoid introducing a variable or constraints for x_t when there have been no witnesses to x_t and it is possible to constrain x_{t+1} directly from x_{t-1}. When a command is introduced, the compiled $\mathcal{M}_\mathcal{T}$ determines what clauses should

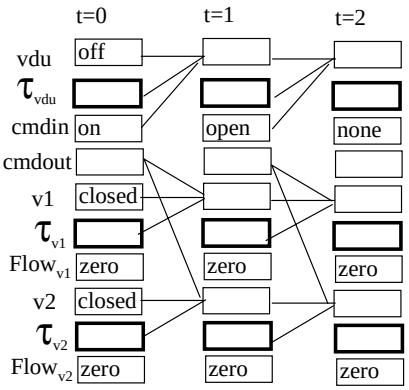


Figure 5.3: Expansion of the VDU Problem to Depth $m = 3$.

be added to constrain the nominal transition of y_t under $\mathcal{C}_{y,t}$. State variables appearing in the introduced clauses are added, along with constraints representing their idle or failure transitions. By reducing the number of variables and clauses introduced at each time step, we reduce the consistency problem involved in checking a trajectory to a number of variables proportional to $m \times \mid \pi_t \mid$. The number of clauses is proportional to $m \times (\mid \pi_t \mid + k)$ where k is the number of failure values per τ_y domain.

5.4 Finite Horizons

While selective extension reduces the variables per time step, we still require an unbounded number of variables over time. We avoid this requirement by setting a finite horizon h steps in the past beyond which all assignments are summarized by a single assignment. We summarize the l most likely assignments to all variables $\tau_{y,t}$ where $t < (m - h)$ into l different assignments to a single variable *History*. All other possible assignments to the initial $\tau_v$ariables are discarded. The horizon point $(m - h)$ is fixed relative to the present, and therefore only a bounded number of variables are required.

While a finite horizon is most useful when m has become large, we will illustrate the concept with a small example. Consider Figure 5.3. The VDU system has been tracked for 3 time steps and the model has been expanded accordingly. The variables $\tau_{vdu,0}$ through $\tau_{vdu,2}$ and $\tau_{v_i,0}$ through $\tau_{v_i,2}$ have been introduced to represent choices in the system's evolution. While tracking the system, we have incrementally generated the most likely consistent trajectories, represented by the most likely consistent assignments to all $\mathcal{T}$. Let us consider the case where the most likely trajectories given the observations thus far have been determined to be a VDU failure at $t = 0$ or a double valve failure at $t = 1$. These are represented by the following assignments:

$$\{\tau_{vdu,0} = \mathbf{Hang}, \tau_{vdu,1} = nom, \tau_{v1,t} = nom, \tau_{v1,1} = nom, \tau_{v2,t} = nom, \tau_{v2,1} = nom\} \quad (5.1)$$

$$\{\tau_{vdu,0} = nom, \tau_{vdu,1} = nom, \tau_{v1,0} = nom, \tau_{v1,2} = \mathbf{Stick}, \tau_{v2,0} = nom, \tau_{v2,2} = \mathbf{Stick}\} \quad (5.2)$$

Note that these are the most likely assignments to $\tau_{y,0}$ and $\tau_{y,1}$ given the observations received in the first three steps. Note that each assignment entails a set of values for Σ_2. For each likely full assignment, we can introduce a single variable assignment that summarizes the full assignment. Consider Figure 5.4. At the left of the figure, we have installed assignment 5.1, thus entailing values for the variables at $t = 2$. At the right, we have eliminated all variables at $t = 0$ and $t = 1$ and directly constrained the variables of $t = 2$ from a new assignment, $History = Hang$. If we define Γ(History=Hang) to be equal to the rank of assignment 5.1, then we have an equivalent representation of this trajectory with far fewer variables. Similarly, in Figure 5.5 we represent the trajectory of assignment 5.2 with the assignment $History = 2Stick$. Once we have summarized a sufficient number of trajectories (here two) with unique assignments to *History*, the variables at $t = 0$ and $t = 1$ are discarded.

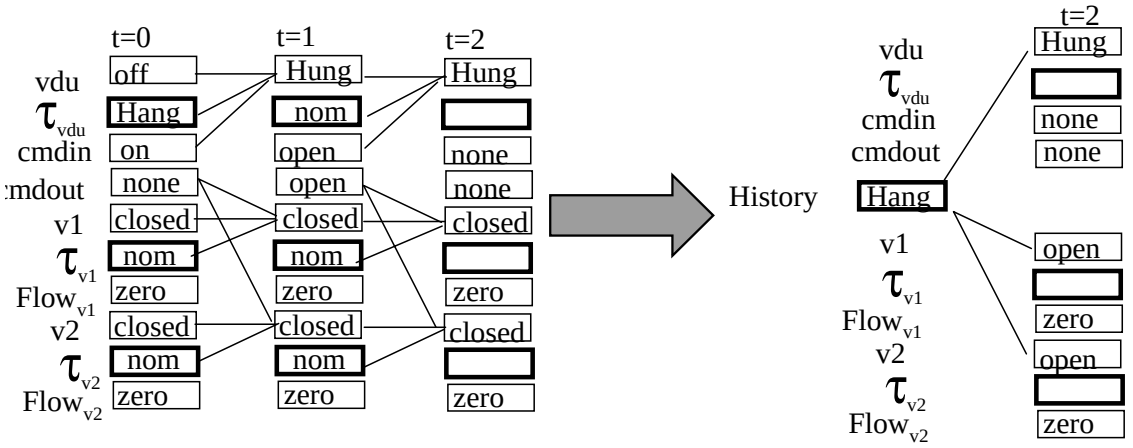


Figure 5.4: Summarization of the initial trajectory, $\{\tau_{vdu,0}$=Hang$\}$

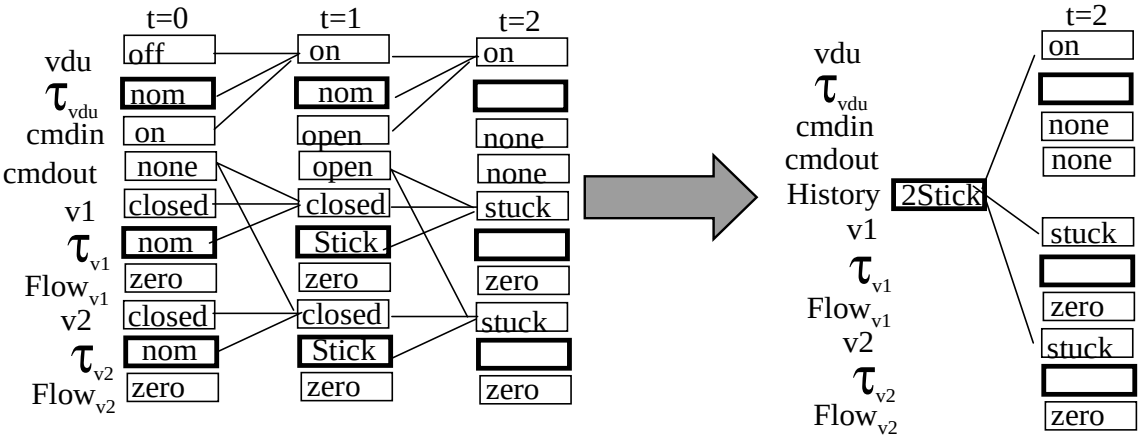


Figure 5.5: Summarization of the initial trajectory $\{\tau_{v1,1}$=Stick, $\tau_{v2,1}$=Stick$\}$.

We need not truncate the horizon after only two time steps. Imagine that we continue to expand the representation for m steps, conditioning on observations and finding the likely trajectories. Each m-step trajectory contains an initial segment that is most likely given a large number of observations received during time steps 0 through m. When m is sufficiently long to give us confidence that our most likely initial trajectories include the actual trajectory, or if we simply can only afford to store m steps, we apply the summarization. We replace each of l likely assignments to $\tau_{y,0}$ through $\tau_{y,(m-h)}$ with an assignment $History = choice_l$ that has the same rank. We then entail the equivalent values in Σ_h. For each value $y_h = y^*$ that was previously entailed by the nontruncated representation, we introduce a clause of the form $History = choice_l \Rightarrow y_h = y^*$.

This summarization may be applied repeatedly. If we expand the summarized representation of Figure 5.4 and Figure 5.5, we may then summarize the l most likely assignments to *History* and all $\tau_{y,2}$ into a new variable *History'*. We may also summarize the oldest time step or several timesteps leading to the oldest time step. By repeatedly applying the summarization as we extend the transition system, we can maintain a fixed size representation. The summary variable $History$ restricts choices for the initial portion of the trajectory to the partial trajectories that appeared most likely after being extended for some time. Intuitively, we are trading the ability to represent an exponential number of initial trajectories with increasingly unlikely prior probabilities for a constant problem size and search space. Unlike the previous approximation, this approximation is not conservative. If the true trajectory involves an assignment to an old $\mathcal{T}$ variable that is not captured in a summarization, it is lost. This case can only arise when a failure too unlikely to be tracked has occurred, yet it was able to remain consistent with the observations until that part of the history was truncated.

Figure 5.6 shows a complete representation making use of both the conservative approximation and a finite horizon. At the right of the figure, new variables are introduced to represent the time steps. Here, where the trajectory assignments have not yet been conditioned on a large number of variables, we have a full model of the system. As variables age, they are moved into a conservative approximation. The assignments here have already been conditioned on observations within their timestep before aging. They will now be conditioned upon how they effect the evolution of the system and whether they maintain consistency with incoming observations. Assignments that have both been conditioned on observations within their timestep and later on how they impact newer observations, and have remained consistent, are summarized into the history variable. Thus the most space and search is reserved for the portion of the trajectory of which we are the least certain.

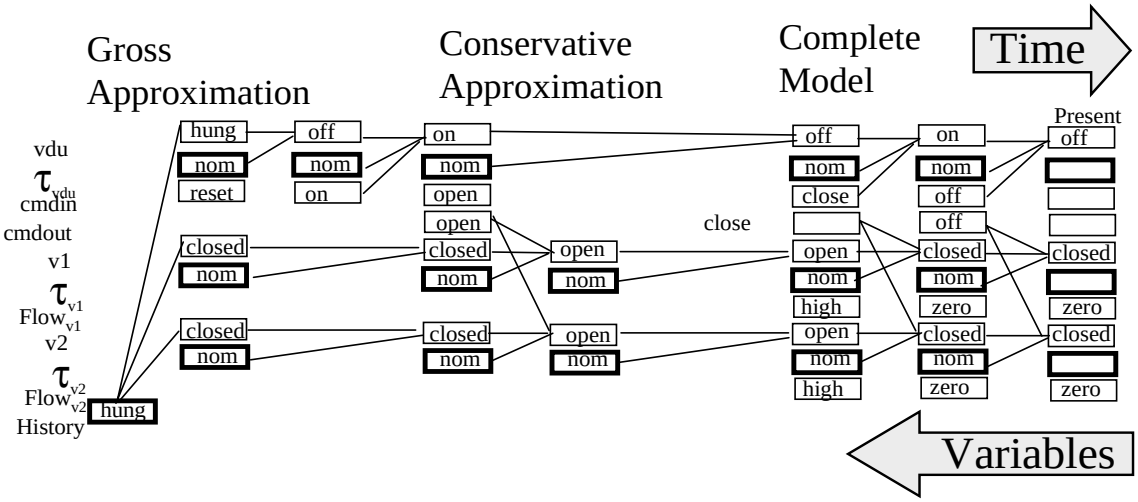


Figure 5.6: The Complete Representation

Chapter 6

Results

We have implemented the transition system representation and the algorithms presented here in a software system called *L2*. *L2* is implemented in C++ in a modular form that allows alternative search and consistency procedures to be plugged into the transition system framework. The tests described below were performed using *CoverTrack* for the search. Consistency was determined via a propositional consistency checker that used unit propagation and a truth maintenance system to cache inferences. Observations were kept for one step only. The growth of the model over time was not limited by a horizon. The tests were run under Windows NT on a 550Mhz Pentium III.

L2 correctly tracks the canonical scenarios known to confound *Livingstone*. Consider Example 3. When the pump is turned on, *Livingstone* finds two conflicts in the current mode assignments: valve v1 cannot be open, and valve v2 cannot be open. It thus fails both valves. *L2* finds the following sets of devices that could not have both transitioned nominally: {VDU, v1}, {VDU,v2}. The lowest cost covering is to fail the VDU at time step 0. Many more interesting scenarios have been demonstrated. If the VDU is failed and v1 is commanded open, the trajectory wherein v1 is stuck will be tracked if that is more likely than a VDU failure. If v2 is later commanded and no flow results, the v1 failure is dropped and the trajectory where the VDU failed in the past is found. If the VDU is resettable, the trajectory wherein the VDU has failed in a resettable manner is first tracked when multiple valves fail to open. If the VDU is reset and the valves again fail to open, *L2* may find a trajectory where the valves were stuck all along, one that replaces the past resettable VDU failure with a permanent failure, or both, depending upon the ranks of the various failures.

Longer runs on more complex models written by *Livingstone* users rather than the authors were also performed. The Circuit Breaker, CB, model of 24 electrical components connected in series and parallel is illustrated in Figure 6.2. This model was tracked in runs of 618 steps. Figure 6.3 illustrates a run wherein every 16 steps the same set of devices are turned on and turned off, a total of 39 times. On the final cycle, a device fails. The CPU times for both the nominal and failure steps are below the clock resolution of

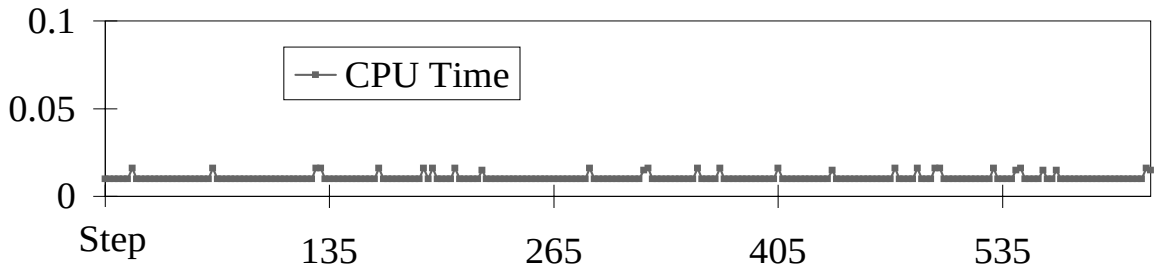


Figure 6.1: CB - Single Failure at Step 598

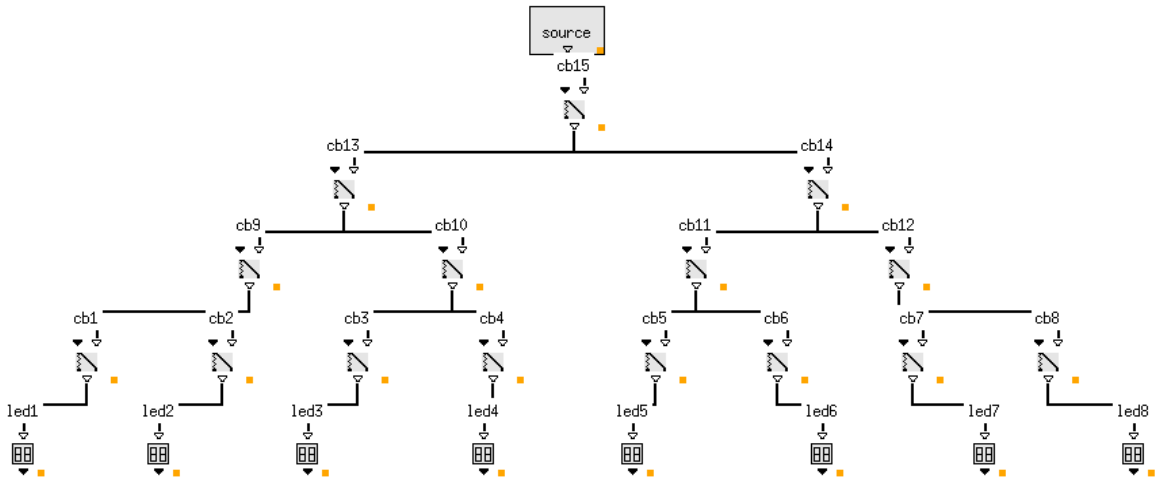


Figure 6.2: The Circuit Breaker (CB) Model

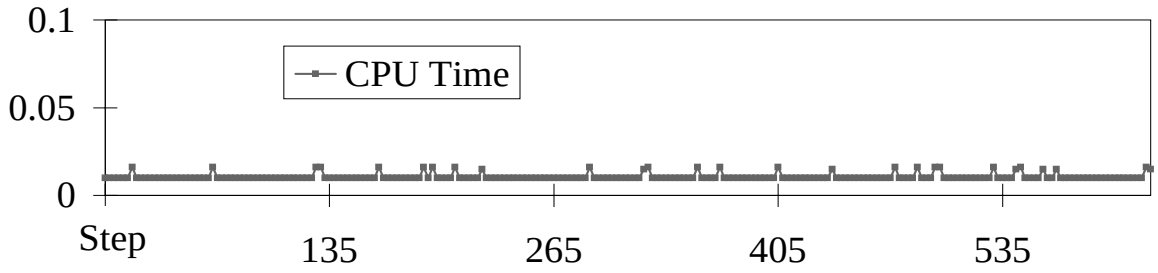
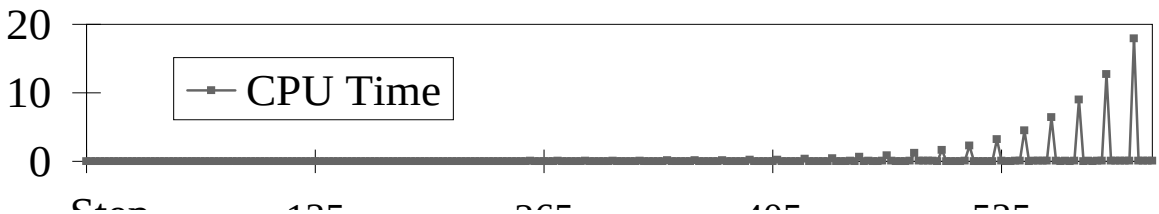


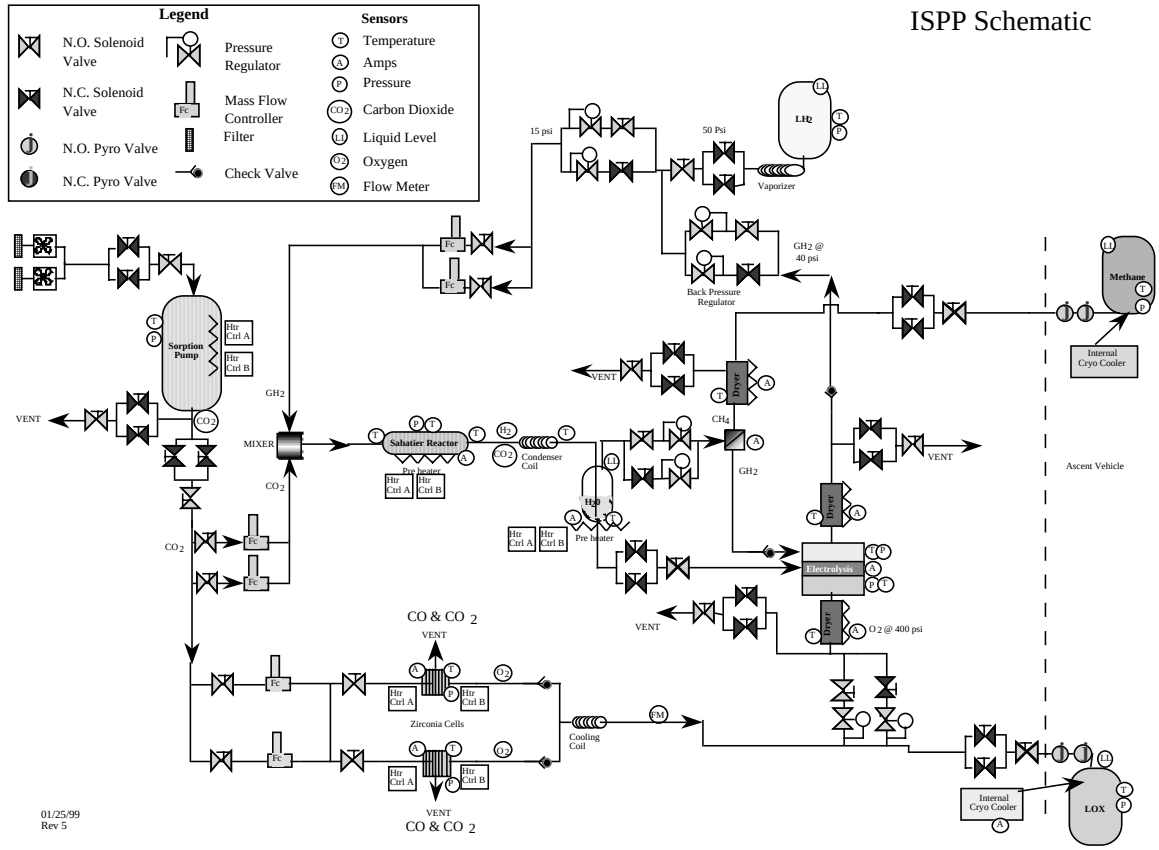
Figure 6.3: CB - Single Failure After 598 Steps

0.016 seconds. Figure 6.4 illustrates a run wherein the device fails on every 16 step cycle and is reset. The device fails a total of 39 times.

The in-situ propellant production model, $ISPP$, has 59 components and represents a chemical processor designed to produce rocket fuel from the Martian atmosphere. The ISPP model was chosen over other, larger models because its *flow failure* requires far more time for diagnosis under *Livingstone* than any other scenario we have encountered. Figure 6.5 illustrates the ISPP hardware, though the components in the upper right quadrant have not yet been modeled in Livingstone. Figure 6.6 illustrates a 33 step track of the model, approximating one day's worth of commands. On the 27^{th} step, the flow failure becomes observable. On steps 32 and 33 simpler, unrelated failures occur. Figure 6.6 illustrates a second tracking run of the same model. Note that the time axis is logarithmic. On step 15 the flow failure is introduced. Repair actions are taken and the failure is immediately reintroduced, until a total of four identical failures have occurred. Additional runs were made on both models. Our results suggest the following.

- *Model growth per time step is small.* ISPP begins at 2933 clauses and grows by an average of 36 clauses per time step. CB begins at 1126 clauses and grows by an average of 44 clauses per time step. *Tracking time steps where no failure occurs takes a very small amount of CPU time.* Note that in Figure 6.3 and Figure 6.6 the steps before the first failure occurs require a negligible amount of CPU time. The nominal steps after the failure in Figure 6.6 take slightly more time, as 8 trajectories are being tracked, but the cost is still negligible.
- *Keeping a history does not induce an unreasonable cost when diagnosing a single failure.* When the nominal trajectory is ruled out we have a single, long conflict and $\gamma = 1$, leading to a simple coverage





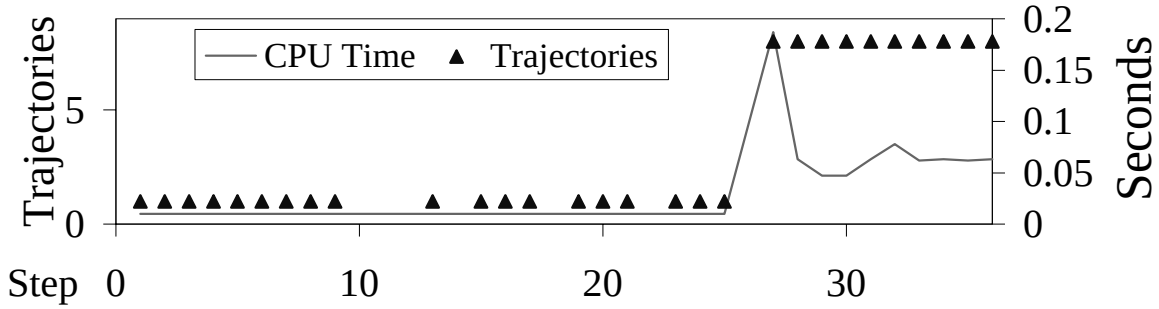


Figure 6.6: ISPP - Independent failures at steps 27, 32 33

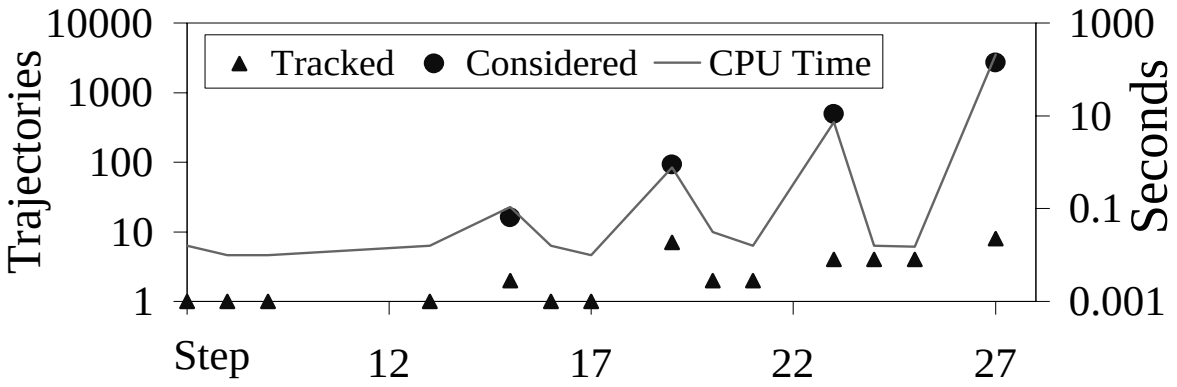


Figure 6.7: ISPP - 4 Identical failures over 27 Steps

problem. The CPU time for a single CB failure scenario is below clock resolution whether 15 or nearly 600 nominal steps, as in Figure 6.3, precede the failure. In Figure 6.7, *L2* finds the eight trajectories that explain the failure that became observable on step 27 in 0.19 seconds.

- *Because of the accumulation of conflicts, tracking the system through k failures spread over time can be an easier problem than diagnosing a single failure of cardinality k.* Consider the run of Figure 6.6. On step 27, the flow failure occurs, causing the large spike in CPU time. Eight trajectories result and are tracked until step 32. On step 32, a simpler, unrelated failure occurs, and none of the 8 trajectories is consistent when extended by the nominal transition. Note that *L2* must now re-diagnose the entire history of the system including the flow failure. It does so in just 0.08 seconds, less than half of the time required to diagnose the flow failure alone. The key to this behavior is the conflicts. On step 27, the nominal trajectory is ruled out and $ConflictDB$ contains a single conflict. *GenerateCover* returns 28 candidate trajectories, 20 of which are ruled out, adding another 15 candidates to $ConflictDB$. Calling *GenerateCover* with the same γ on these conflicts almost immediately returns just the 8 consistent candidates. On step 32, the 8 diagnoses are ruled out by conflicts resulting from the simple failure. Since the conflicts from the simple failure involve none of the variables from the flow failure conflicts, the problem decomposes into two subproblems, one of which has previously been solved and memoized by the conflicts.
- *Unfortunately, tracking k related failures over time can also be as computationally intensive as diagnosing a cardinality k failure.* Figure 6.7 illustrates a sequence of failures where the conflict coverage problem does not decompose. At each time peak, the flow failure has occurred again. The conflicts generated by the fourth failure involve exactly the devices involved in the first three failures. As a result, the time required to solve the hitting set problem and the number of inconsistent trajectories considered rises dramatically.
- At the fourth failure, 2694 candidates are returned in 174 seconds. An additional 33 seconds are spent determining all but 8 of them are inconsistent. Figure 6.4 clearly shows the exponential growth of tracking time as the number of failures involving the same device grows.

Chapter 7

Related Work in Acting Under Uncertainty

The preceding chapters explore methods for determining a partial distribution over the possible state an apparatus occupies. Given that the control system no longer knows with certainty what state the apparatus occupies, the question arises of how does one choose actions? An action that achieves some desirable goal when the apparatus is in one possible state may be ineffective or precipitate a disaster if the system if the apparatus is in fact in another state. The traditional method for modeling this state of affairs is to create a reward that expresses the relative desirability of taking an action in a state. Igniting an engine in a state where thrust is desired is assigned a positive reward. Igniting an engine in a state where there is a fuel leak is assigned a large negative reward. Given the reward assignments for each action in each state and a probability distribution over the states, we can use notions from decision theory to determine the utility of each state and choose an action.

Recall that our discrete control problem can be modeled as a partially observable Markov decision process, or POMDP. A policy is a mapping from belief states to actions. Given any belief state, the policy specifies an action to be taken. To solve a POMDPis to create a policy that for any distribution over the state space returns the optimal action to take in order to maximize the expected reward given the partial observability. The details of finding an exact solution to a POMDPare beyond the scope of this proposal but a number of algorithms for finding such a solution exist (Sondik 1971), (Cheng 1988), (Littman, Cassandra, & Kaelbling 1995). Unfortunately, even the most efficient of these would do well to solve problems with 10 states and 10 observations.

Fortunately, a wide variety of technique for more limited versions of POMDP solutions have been developed. These techniques typically involve reducing the problem complexity by making simplifying assumption about uncertainty (*e.g.*, assuming the world is observable or has deterministic actions), generating something other than a policy that maximizes expected reward from any state (*e.g.*, assuming there is a single goal state that must be reached) or some combination. Figure 7.1 shows one possible hierarchy of the methods, roughly categorized by how the techniques deal with uncertainty. The MDP technique generates a full policy, applicable in any state, but assumes the current state is always observable. Conformant planning techniques assume the initial state is unobservable but contained within a small set, and generate a plan that succeeds (possibly with some probability) regardless of the initial state. Contingent planning techniques generate a branching plan whose branches are chosen at execution time based upon the results of observations. Belief replanning makes an initial assumption about the start state or initial distribution and creates a deterministic plan appropriate to that assumption. If at any point the belief state predicted by simulating the deterministic plan diverges significantly from the belief state resulting from execution of the plan, either the initial assumption was incorrect or the plan did not behave deterministically. In either case, a new plan is generated from the divergent belief state. The remainder of this chapter describes these methods and their applicability to the problem at hand.

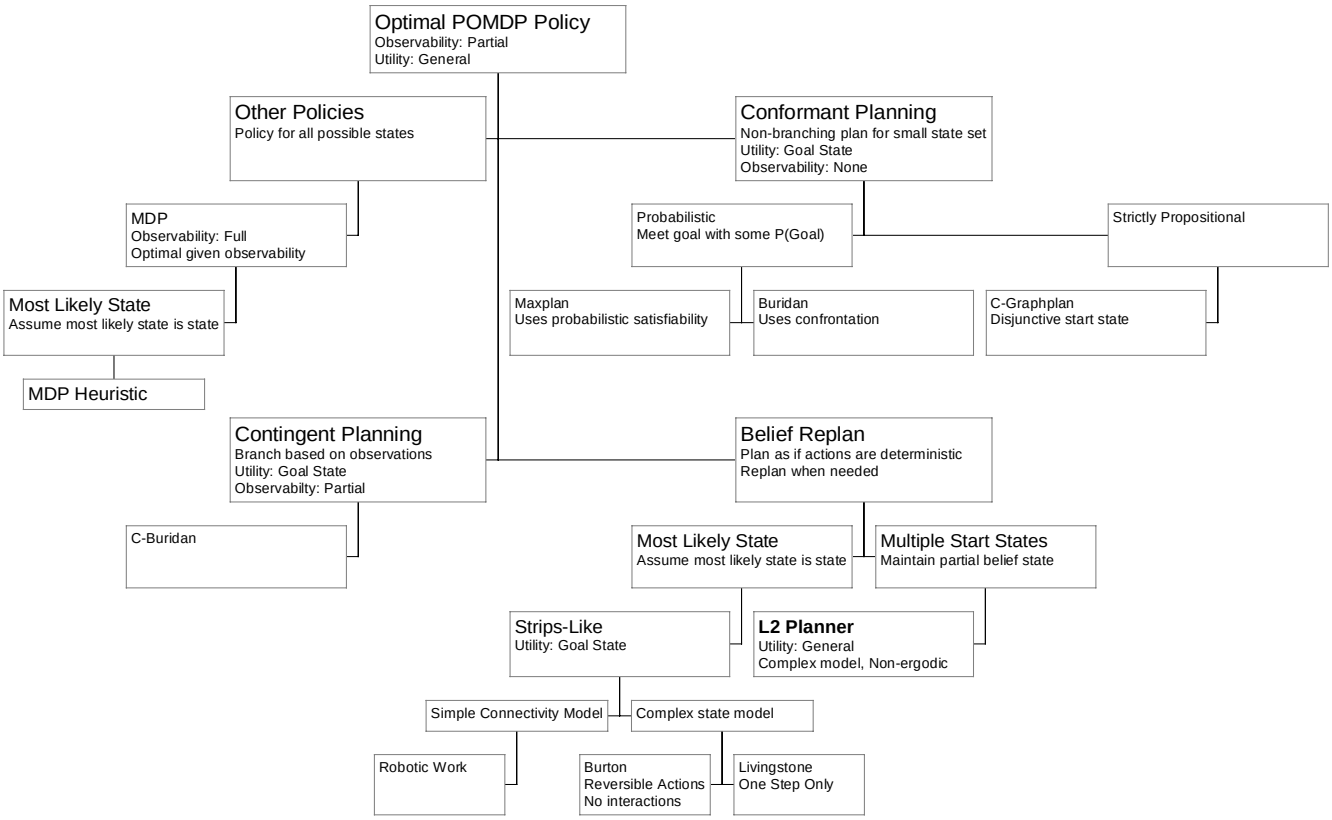


Figure 7.1: A Hierarchy of Planning Problems

7.1 Policy-based Solutions

7.1.1 MDP-based Heuristics

A POMDPwith deterministic observations (*i.e.,* each observation reveals the true state of the apparatus) is a Markov decision process, or MDP . Given complete observability, the exact solution to an MDP is a policy that specifies the optimal action to take given that the outcome of an action is non-deterministic, but will be known once the action is taken. Such a policy can be obtained quickly even for relatively large models.

In MDP -based POMDPalgorithms, a suboptimal solution to the POMDPis developed from the optimal solution to the corresponding MDP via the MLS , or *most likely state* assumption. Given the belief state, this heuristic finds the world state with the highest probability. It then uses the MDP policy to select and execute the action that would be optimal if the current state and the state resulting from the action were directly observable. This is of course a heuristic, since the state that is most likely given only the current observations may not be the true state, and the action that is optimal given complete knowledge of the state (*e.g.,* whether or not a fuel valve is leaking) may not be optimal given only a distribution over the possible states.

The strength of the MDP heuristic is that it takes into account the uncertainty in the results of actions, thus penalizing actions that might take the system into or near an undesirable state. This strength is undermined by the infinitesimal interpretation of the transition system: the nominal transitions dominate to the extent that non-deterministic outcomes of the action are only considered when conditioning on observations forces it. In addition, as failures are modeled as occurring from any state, it's not clear how one would act in order to avoid precipitating failure. Thus the main strength of the MDP heuristic, the ability to take into account possible unexpected outcomes of an action before there is an evidence to suggest they may have occurred, seems wasted in this framework. Perhaps more importantly, this heuristic requires solution of the underlying MDP . While an MDP with tens of thousands of states can be solved, the state space of the models we seek to operate on precludes explicit solution of the corresponding MDP .

7.2 Maximizing Expected Reward

Even if we cannot compute a full policy, we might still take advantage of the general utility model of an MDP by attempting to find a partial policy or plan that attempts to recommend the actions that maximize the expected reward over some horizon. This has a few complications. If we assume the only information we have about the prior probability of the outcome of an action is an infinitesimal rank, computing the expected reward of an action is not trivial. One simply cannot multiply a rank and a reward. One approach is to compare expected rewards in a rank-wise, lazy fashion without ever explicitly computing an expected reward value. That is, if we wish to compare the expected reward of plan $p1$ and plan $p2$, we first compute the reward that would be received in the rank 0 (no failure) execution of the plans. If one clearly dominates, we stop. If one does not dominate, we compute the reward that would be received over all executions of the plan of the next rank. This lexicographical comparison of plans allows lazy evaluation, and avoids the need to take the product of probabilities (specified as ranks) and costs (specified as real numbers).

Unfortunately, this approach does not have the needed expressiveness. Consider again the problem of the computer that can be frozen due to a software hang or a hardware hang. A reset fixes the software hang, does nothing to a hardware hang, and never causes additional failures. Power cycling the hardware fixes both the software and hardware hangs, but sometimes destroys the hardware. Looking only at the rank 0 outcomes, power cycling dominates. If our initial distribution contains both types of failures, power cycling repairs both while resetting repairs only one. However, power cycling in this situation is often not the preferred action. A reset is often chosen because while in one case it does nothing, in both cases it

does no harm. Choosing between these actions requires some analog of summing the expected reward of the differently ranked outcomes of each plan in order to determine which has the highest total expected reward. In addition, it's not clear how to compute the expected reward for a sequence of actions. A plan might fail at any point. After this failure, the remainder of the plan might have a considerably different effect than was planned. This must be accounted for in computing the expected reward for execution of the plan. Note that we are monitoring execution of the plan using our state estimation techniques and will stop execution as soon as it is noticed that the plan execution has gone wrong. Thus the length of the remaining plan that is actually executed, and will actually result in reward, is not clear.

We could attempt to address these issues by determining an algebra for combining rank likelihoods and costs, perhaps by using rank costs, and by using the worst case reward or reward up to the first failure to approximate the reward received for a plan execution. However, we have chosen to initially sidestep these issues until we have explored solutions that avoid them.

7.3 Belief Replanning

Belief replanning occurs at the opposite end of the uncertainty spectrum as an optimal policy. This technique creates a deterministic plan from the current state to the goal state, and replans when it's clear something has not gone as expected. The algorithm starts by assuming it is in the most likely world state and that a deterministic idealization of the action model holds. In this idealization, the most likely outcome of an action in a state is assumed to be the outcome that always results from performing that action in that state. Given these assumptions, the algorithm generates a deterministic plan from the most likely state to the goal state along the appropriate deterministic actions. In addition, it generates a sequence of predicted world states that will be traversed if this deterministic trajectory is followed. The system then embarks upon this plan, updating its belief state and checking, at each step, that the most likely world state (according to the belief distribution) is equal to the predicted state. If it is not, the cycle begins again by planning from the current most likely state.

This family of approximations has been used in mobile robots (*e.g.*, Dervish (Nourbakhsh, Powers, & Birchfield 1995)) and is also very similar to the methods developed independently in the model-based diagnosis world and embodied in the Livingstone and Burton systems (Williams & Nayak 1996).

Since belief replanning plans using a simplified deterministic action model, it will not take into account less likely outcomes of an action that result in negative reward or disaster as the MDP -based solution will. Given the strong bias of our models to the nominal outcome of actions, this may be one reasonable approach to avoid solving the underlying MDP .

The following sections detail a number of variations.

In fact, the next two subsections describe specializations of belief replanning that we have used in practice to perform action-selection on very large transition system problems.

7.3.1 Livingstone

With the MLS assumption and the assumption of deterministic actions, belief replanning is significantly simpler than solving a POMDP, and has been successfully applied in realistically-sized robot navigation domains (Nourbakhsh, Powers, & Birchfield 1995), (Cassandra, Kaelbling, & Kurien 1996). In the robot domain, the action model typically models the topology of a physical space. The action model is a matrix where T(s,a,s') non-zero if s' is reachable from s by performing action a. When the action model is assumed to be deterministic, the planning problem is reduced to the polynomial time problem of finding the shortest path between nodes in a graph (Dijkstra 1959). In the transition system representation and most traditional planning domains, the state space is too large to allow us to explicitly express the transition model. T(s,a,s') is therefore implicitly encoded in a logical action model that includes a propositional

precondition that must be entailed in order for an action to be enabled. In addition, the goal state is not specified directly but rather constrained by a propositional formula it must entail. Thus, even if the belief replanning assumptions are made, action selection remains full STRIPS-style planning.

The reconfiguration algorithm used in Livingstone (Williams & Nayak 1996) is a variation on belief replanning designed to work with a implicit, propositional specifications of T(s,a,s') and the goal state, and to provide sub-second response times. Livingstone avoids solving an unrestricted STRIPS planning problem and draws upon existing techniques in model-based diagnosis by assuming that the plan required to reach a goal state consists of a single set of actions performed in parallel. Livingstone searches for a set of actions for the components of the apparatus that, when executed in parallel and assuming determinism, move the system from the current most likely state to a state that entails the goal specification. Considering a only single step greatly simplifies the planning problem, allowing issues such as re-ordering steps to avoid clobbering of preconditions to be ignored.

Livingstone casts the one-step planning problem as search for an assignment to the modes of the components of the system that entails the goal specification. Livingstone performs this search using the same conflict directed, best first search over mode assignments used in diagnosis. During diagnosis, Livingstone searches for an assignment to the mode variables that is consistent with the observations, guided by conflicts between observations and the predictions entailed by the current mode assignment. Mode assignments that support entailment of a prediction that conflicts with an observation are candidates for reassignment to a failure value.

During action selection, Livingstone searches for an assignment to the mode variables that entails the goal specification, guided by conflicts between the desired state and predictions entailed by the current mode assignment. Mode assignments that support a prediction that is in conflict with the desired state must be reassigned. A mode is reassigned by finding an action that moves the mode from the current to a new assignment. Sets of reassignments to the conflicting modes are considered in best first order, using as a metric the sum of the costs of the actions required to make the reassignment. The search terminates when a mode assignments is found that entails the goal specification. The set of actions used to reassign the modes to the entailing assignment is returned as a one-step parallel plan.

This technique has been shown to very rapidly find single-step plans for complex domains and has used in the control of a spacecraft (Bernard *et al.* 1998) and in several other applications (Kurien, Nayak, & Williams 1998). It's limiting weakness is that not every goal state can be reached using a single step plan. For example, the simple act of power cycling a device in the hope of resetting it cannot be represented. In practice, critical sequences of sub-actions can be made into a macro action that is added to the model. For example, a ıpower-cycle action that reassigns a component's mode from ıhung to ıon was used as a workaround in the spacecraft application. The system executing the actions was then made responsible for translating the ıpower-cycle action into a sequence of turning the device off then on.

Unfortunately, this workaround can add a great deal of complexity to whatever system is executing the plan returned by Livingstone, as it must decode each macro action into a sequence and potentially interleave the sequences resulting from multiple parallel macro actions without producing any negative interactions.

7.3.2 Burton

The Burton model-based reactive planner (Williams & Nayak 1997) is a further refinement of belief replanning that complements Livingstone. It eliminates the need for macro actions by eliminating the single step assumption. Burton finds a sequential plan that leads from the current most likely state to a goal state. In order to generate a candidate goal state from the goal specification, Burton uses Livingstone's recovery algorithm, described above, to find a set of goal mode reassignments that entail the goal specification.

Burton then finds a plan that moves the current modes to the goal mode assignment. By leveraging a set of simplifying assumptions and preprocessing the system model, Burton can return the first step in this plan in average case constant time. Burton is therefore used by generating the first step of the plan, executing it, then replanning from the new belief state. Note that while we generate and execute the plan one step at a time to maximize response time and robustness to unexpected events, each step is guaranteed to be the first step in a sequential plan to the goal state.

To achieve it's performance, Burton assumes it's possible to avoid negative interactions between mode reassignments. That is to say, given a set of modes that need to be reassigned, one can perform the actions to reassign a mode variable without undoing any reassignments that have been done, and without making any remaining reassignments impossible. For example, in order to change the mode of the valve we must change the mode of the valve driver. However, once we achieve the desired mode of the valve, the valve is unaffected by further changes in the valve driver's mode. Thus we plan reassignment of the valve before final reassignment of the valve driver to its goal assignment. This assumption is not unreasonable for many engineered devices assembled from independent components. It requires that

- There are no irreversible actions
- Every device is independently commandable.
- Every device has an idle command (or the transmission of no command) that causes the device to persist in its current mode
- Device is tree structured in terms of commanding

Note that the final requirement does not preclude the existence of continuous feedback loops within components of the device. However, issuing a mode-changing command to a component cannot result in a second command being issued to the component via a command feedback loop.

If these requirements are met there will exist a topological ordering of the mode variables such that if component $C1$ appears before $C2$ in the ordering, the plan necessary to put $C2$ in its desired mode is not dependent upon nor does it change the mode of $C1$.

Definition 7 A component $C2$ is said to be *upstream* of component $C1$ if the transition clauses that model the evolution the state of $C1$ involve $C2$ (*e.g.,* a communication bus needs to be on in order to command $C2$). $C2$ must appear after $C1$ in the topological ordering.

The topological ordering and these restrictions ensure that $C2$ is reconfigured before all of its upstream devices. We are therefore free to command the upstream devices as needed. When the upstream devices are later moved to their goal configurations, $C2$ can be held in its goal configuration. Thus any reconfiguration problem is reduced to a collection of small reconfigurations of a single device and its upstream devices. In order to further decrease response time, Burton does not generate a complete plan to reconfigure the system, but rather returns the first step in a plan from the current state. Note that Burton is guaranteed to return the correct first step in a multi-step plan leading from the current MLS to the goal state, is as opposed to Livingstone, which returns a single step plan. If finding the first step to a plan can be made fast enough, Burton's approach is more efficient than standard belief replanning, in that it never generates a plan sequence that is thrown away because of non-determinism.

Burton achieves this speed by compiling a policy off-line that for any current and desired mode of component C, specifies the action to perform. This policy is only applicable when the upstream components are in the appropriate modes to allow C to be commanded. If the upstream devices are in the correct modes, the policy is used to read out the appropriate action to reconfigure C. If the upstream components are not in the correct modes, their upstream components and policies are similarly checked to read out the action that must be taken before reconfiguring C. In either case what is returned is the first step in the plan

to reconfigure C. The depth of the chain which can be followed before looking up an action is bounded by the topology of the device, leading to an average-case constant time reactive planning algorithm.

7.4 Inapplicability of MLS-based approaches

There are a number of features of our domain that limit the utility of the MLS -based approaches. Most fundamentally, there is quite often no most likely state. The infinitesimal interpretation of the transition priors, or any other order of magnitude probability specification that is likely to be required in practice, often results in several states being assigned the same relative probability. Doing something more intelligent than choosing a state at random requires acknowledging the initial distribution over the start states. This in turn requires acknowledging that when the initial state is not known with certainty and an action is taken, the resulting state will not be known with certainty. Thus the very question of finding a sequence of actions that reaches the goal state (with implicitly certainty) must be revised. Second, we often care about the path taken to the goal configuration. For simplicity, belief replanning, Livingstone and Burton all assume that the agent is rewarded for entering the goal state and receives no reward, positive or negative, in any other state. Thus there is no way to model the undesirability of action sequences that take the system through undesirable or dangerous states on the way to reaching the goal state.

Chapter 8

Safe Planning

As discussed in the previous chapter, a simple STRIPS-like framing of the planning problem, with a single start state, a goal state, and no costs related to performance of actions, is inadequate for capturing the type of capabilities we would like our reactive planner to have. The full POMDPproblem is certainly general enough, but is intractable. We require a specification of the planning problem that is more general that the STRIPS framing, but remains tractable. Intuitively, given a small set of possible current states, we would like a sequence of actions that reaches a state that meets some goal specification. Of course, since we do not know which of the current states is the true state, and some may include irrecoverable failures, we cannot guarantee that a single plan will reach the goal from all possible current states. We would therefore settle for a plan that reaches a goal state from at least one of the current states. Of course, there may be many such plans, and not just any of them will do. A plan that when reaches the goal state from one possible current state and does nothing in all other states is preferable to a plan that reaches the goal state from one possible current state and reaches a highly undesirable state from possible current states. Similarly, a plan that reaches the goal while avoiding passing through any undesirable states is preferred to one that enters undesirable states before reaching the goal. We will capture the notion of undesirable states through the use of a set of safety constraints that mark a subset of the state space as unsafe. We will then define several varying notions of what it means for plan to be safe. Similarly, we do not want a plan that needlessly uses resources or reduces our flexibility in reaching future goals. We will capture these notions by giving a cost to taking actions and marking a subset of the actions that reduce our future flexibility as irreversible. From these we will define a notion of the optimality of a safe plan. This chapter only describes the formulation of the safe planning problem. Development of algorithms for the problem are future work.

8.1 Compilation of the Transition System for Planning

In order to clarify the planning problem and algorithms for addressing it, we would like to simplify our transition system representation to only those components required for planning. This would include a set of state variables, actions on those variables, safety constraints on those variables, and a goal predicate upon those variables. Specifically, we would like to eliminate the dependent variables and the state constraints from the model. This will allow us to characterize the problem as an extension to STRIPS-style planning.

8.1.1 Eliminating the State Model, $\mathcal{M}_\Sigma$ and $\mathcal{D}$

A transition system is a tuple $\langle \Sigma, \mathcal{T}, \mathcal{D}, \mathcal{C}, \mathcal{M}_\Sigma, \mathcal{M}_\mathcal{T}, \Gamma \rangle$, where Σ represents the controllable state of the system, $\mathcal{T}$ and Γ represent non-deterministic choices over actions, $\mathcal{D}$ and $\mathcal{M}_\Sigma$ determine if an assignment to Σ is consistent with observations, and $\mathcal{M}_\mathcal{T}$ models the transition of the value of Σ from one time

point to the next. We first eliminate $\mathcal{M}_\Sigma$. The purpose of $\mathcal{M}_\Sigma$ is to determine whether an assignment to Σ is consistent with observations[1]. In the planning problem, there will be no observations. $\mathcal{M}_\Sigma$ is therefore trivially consistent regardless of the assignment to Σ and can therefore be eliminated. Recall that $\mathcal{M}_\mathcal{T}$ can be compiled so that all references to $\mathcal{D}$ are eliminated, yielding a formula on $\mathcal{C}$, Σ and $\mathcal{T}$. This eliminates all remaining references to the variables of $\mathcal{D}$, which is in turn eliminated.

8.1.2 Eliminating $\mathcal{T}$ and Translating $\mathcal{M}_\mathcal{T}$ to STRIPS Actions

During planning, we will only be considering the most likely outcome of each action. We may therefore discard any constraint from $\mathcal{M}_\mathcal{T}$ that mentions any but the most likely assignment to each variable $\tau_{y,t}$. Each remaining constraint is of the form

$$\mathcal{C}_{y,t} = c' \wedge \pi_t \wedge (\tau_{y,t} = nominal) \quad \Rightarrow \quad y_{t+1} = y''$$

where π_t is a formula involving only Σ_t. As we are only considering the case where each ($\tau_{y,t}$=nominal), we may we may eliminate $\mathcal{T}$ and Γ. We may also split π_t into a form that makes explicit the dependence upon y_t and some other state variables x_t through z_t.

$$\mathcal{C}_{y,t} = \mathcal{C}^* \wedge y_t = y' \wedge x_t = x' \ldots \wedge z_t = z' \quad \Rightarrow \quad y_{t+1} = y''$$

For each such formula remaining in $\mathcal{M}_\mathcal{T}$, we can introduce an equivalent STRIPS-style action A with the preconditions $\{\mathcal{C}_y = \mathcal{C}^*, y = y', x = x' \ldots z = z'\}$ and effects {add y=y", delete y=y'}. (Note this translation assumes the precondition is a conjunction, though a formulation with any WFF is similar. This needs to be addressed.)

8.2 Formulating the Safe Planning Problem

Our basic problem structure is a disjunctive start state with a goal predicate and safety constraints. Intuitively, we would like, when possible, the system to reach the goal while entering only states that satisfy all safety constraints, but will accept plans that do not contain actions that violate a constraint that was not violated in the previous state. We might also like to formulate the problem such that we are allowed to temporarily violate some constraints to reach the goal, but initial states that do not reach the goal may not become worse. We describe each of these planning variations, beginning with some supporting definitions.

- Let $\mathcal{S}$, be a uniform distribution over a set of possible initial states. Each member of $\mathcal{S}$, is an assignment to Σ.
- Let $\mathcal{A}$ be the set of possible actions, and $\mathcal{C}(a)$ be the cost of taking an action.
- Let $Goal(s)$ be a predicate on the assignments to Σ that determines if assignment s is a goal state. Note that we may specify a goal as a predicate on $\Sigma \cup \mathcal{D}$ then use $\mathcal{M}_\Sigma$ and prime implicate generation to yield a predicate on Σ.
- Let $\mathcal{Q}$ be a set of predicates $\{q_0 \ldots q_n\}$ upon assignments to Σ. $\mathcal{Q}$ is the set of safety constraints. Note that we may specify a safety predicate on $\Sigma \cup \mathcal{D}$ then use $\mathcal{M}_\Sigma$ and prime implicate generation to yield a predicate on Σ.

Definition 8 Given a start state s, a plan p is **safe** with respect to s if execution of p does not violate any safety constraints not violated by s. That is, $\forall q_i \in \mathcal{Q}$ if s_j precedes s_k in the deterministic execution of p on s, then $q_i(s_j) \Rightarrow q_i(s_k)$. Let this be denoted by $Safe_p(s)$.

Definition 9 Given a plan p and an initial state s, we define the predicate $Goal_p(s)$ to be true if deterministic execution of p starting in state s results in a state that satisfies $Goal(s)$.

[1] We disallow the case where $\mathcal{M}_\Sigma$ contains constraints directly between members of Σwithout including $\mathcal{D}$.

Definition 10 Given a start set $\mathcal{S}$, a plan p is a **maximally safe plan** if p is safe for all start states. That is, $\forall s \in \mathcal{S}$, $\mathit{Safe}_p(s)$.

Definition 11 Given a start set $\mathcal{S}$, a plan p is a **single goal safe plan** if the plan takes at least one state to the goal state, and the plan executes safely from all other states. That is, $\exists s \in \mathcal{S}$, $(Goal_p(s) \wedge \forall s' \in \mathcal{S}, s' \neq s \Rightarrow \mathit{Safe}_p(s))$.

Definition 12 Given a start set $\mathcal{S}$, a plan p is a **goal safe plan** if p either takes a state to a goal, or executes in a safe way from that state. That is, $\forall s \in \mathcal{S}$, $Goal_p(s) \vee \mathit{Safe}_p(s)$.

Definition 13 For the purposes of simplifying the text, we define a **safe plan** as a plan that is maximally safe, single goal safe or goal safe.

Definition 14 The planning problem: Given a start state set $\mathcal{S}$, find the least-cost maximally safe {goal safe, single goal safe} plan.

8.2.1 Discussion

Some interesting things to note:

- Conformant planning is a special case of goal safe planning wherein all start states reach the goal.
- The only uncertainty introduced is in the start state. This frees us from having to compute the expected utility of the execution of a plan.
- Riskier actions are penalized directly by an action cost, rather than computing the expected outcomes of the risky action and using the cost of the outcome to compute expected utility.
- The problem as stated is more sensitive to outcomes along the expected execution than to action cost. That is, no plan may be unsafe. Among the remaining plans, we optimize cost.
- This is basically an improvement to belief replanning. In belief replanning you have some belief about what states you might be in. You arbitrarily pick a state s among the most likely, and plan as if you were in that state. Executing the plan will bring information about the true state. If at some point s is no longer among the most likely, you replan from one of the most likely states. The basic improvement is rather than planning from one state and ignoring the rest, we are a bit more conservative. We plan from one state ensuring that the effects that might occur in other possible starts states are safe.

8.3 Extensions to Safe Planning

8.3.1 An Extension that Distinguishes Irreversible Actions

This planner is aimed at rapidly finding short plans to reconfigure a system. It is not attempting to solve planning problems that involve long-term projection over time. Still, we would like to avoid taking an action during the current planning session that unnecessarily reduces our options for responding to future reconfiguration requests. For example, spacecraft typically employ pyrotechnic devices that can be used once for tasks such as opening the valves to a backup engine. We would like to ensure we use such irreversible action only when warranted and only after proper deliberation. [2]

Definition 15 An action is irreversible if when executed in state s, there exists no other action or sequence of actions that returns the system to state s.

Let $\mathcal{A}_\nabla$ be the set of actions that are irreversible.

[2] Issue: Dave Smith suggested we start with the more general notion of damaging actions, then specialize to irreversible actions, which have algorithmic implications. How do we distinguish the two, in that if damage is not irreversible, we can just undo it.

We require that $\forall a_i, a_j \in \mathcal{A}, a_i \in \mathcal{A}_\nabla a_j \notin \mathcal{A}_\nabla \Rightarrow \mathcal{C}(a_i) \gg \mathcal{C}(a_j)$.

That is, the cost of irreversible actions dominate the cost of reversible actions. The cost of any plan with one or more irreversible actions is greater than any plan without any irreversible actions. This has some potentially interesting ramifications.

- We require that irreversible actions dominate the cost. As a result, we can divide the cost of a plan into an irreversible part and a reversible part.
- We can potentially, off-line, devise a policy that tells us when an irreversible action must be taken based on features of the current state and the goal.
- If no irreversible actions are required, we can solve the problem as a fast reversible-action-only planning problem and know we have the lowest cost plan.
- If a reversible action is required, we can use the off-line policy to take the action with confidence without requiring a large amount of computation. Ideally we'd like to perform the irreversible actions (and whatever reversible setup actions are required) and then be left with a completely reversible problem. We can potentially solve that quickly.

8.3.2 Urgent States

Now we have a planning problem that includes multiple start states, costs on actions, and irreversible actions, which were the top items on our wish list. It requires that we find a plan that reaches the goal from some arbitrary state s from $\mathcal{S}_\prime$, and makes the remaining states no worse. In executing this plan, the idea is we either make progress toward the goal from s, or we make progress toward discovering that s was not the true state. In a practical sense, we can improve it by choosing more carefully from $\mathcal{S}_\prime$. We must choose some s_i to plan from and the remaining s_j will potentially found to be true as a side effect. Note that there may be some urgent states in the distribution, which if they were the actual state, we wouldn't want to discover that serendipitously some time in the future. Instead, we'd want to plan as if that was the state, preferring the possibility that we later discover the true state was a less urgent state. To accomplish this, we add an urgency measure.

- Let $\mathcal{U}$(s) denote the urgency of state s.

Definition 16 The urgent planning problem: Given a start state set $\mathcal{S}_\prime$, find the least-cost safe plan P such that $\exists s \in \mathcal{S}_\prime$ for which P executed on s results in a state that satisfies the goal predicate, and if there $\exists s_j \in \mathcal{S}_\prime$ for which there is also an safe plan to the goal, $\mathcal{U}$(s) $\geq \mathcal{U}(s_j)$.

If we cannot reach the goal from the most urgent state, we might take the next most urgent state, as above, or we might want to eliminate the urgency. Below we give two possible problem statements.

- Let s_f denote the system safe state.

Definition 17 The urgent planning problem with a safe state: Given a start state set $\mathcal{S}_\prime$, find the least-cost safe plan P such that $\exists s \in \mathcal{S}_\prime$ for which P executed on s results in a state that satisfies the goal predicate or results in the safe state if there is no path to the goal. It must be the case that if there $\exists s_j \in \mathcal{S}_\prime$ for which there is also an safe plan to the goal or a safe state, $\mathcal{U}$(s) $\geq \mathcal{U}(s_j)$.

Definition 18 The urgency reduction problem: Given a start state set $\mathcal{S}_\prime$, find the least-cost safe plan P such that $\exists s \in \mathcal{S}_\prime$ for which P executed on s results in a state s_k that satisfies the goal predicate or for which $\mathcal{U}(s_k)$ is less than some threshold. It must be the case that if there $\exists s_j \in \mathcal{S}_\prime$ for which there is also an safe plan that reaches the goal or reduces urgency, $\mathcal{U}$(s) $\geq \mathcal{U}(s_j)$.

8.4 Potential Techniques for Safe Planning

Our task is to solve the safe planning problem: given a set of possible initial states, find a safe plan. Note that this is a generalization of conformant planning, in that we may adjust the number of initial states that must reach the goal from one to all (the conformant case) by modifying the definition of safety. There are two approaches to generating a safe plan, which we will call safe generation and repair. In safe generation one conceptually plans in all possible current states at once. One chooses an action that does not violate any safety constraints no matter which of the possible initial states is the actual initial state. That action, when executed on the possible initial states, results in a set of possible second states. One must then choose an action that does not violate any safety constraints regardless of which of the possible second states is the true second state. This process repeats until one or more goal states are in the set of possible current states. With repair, one uses a traditional planning system to generate a plan that reaches the goal state from one of the possible initial states. There is of course no guarantee that this plan is safe when executed from any of the other possible states, so its effects from those states must be simulated. If a violation of a safety constraint is found, the violation is analyzed to find a ”repaired” plan that does not exhibit the violation. The repaired plan is then simulated and any additional constraint violations are repaired.

While safe generation or repair may lead to an efficient algorithm for safe planning, we have chosen to begin experiments with a repair-based approach. We begin with the Blackbox planner (Kautz & Selman 1999), a fast planner that compiles a traditional planning problem of reaching a goal from a single start state into a propositional satisfiability problem. We first trivially extend Blackbox to generate a plan to reach the goal from a start state S it chooses from a set of possible initial states. If we could ensure the resulting plan did not violate any safety constraints if it were executed in a state other than S, we would have a method for generating goal safe plans. The approach for achieving this is as follows: given the initial plan from our extended Blackbox, we use the existing inference machinery of L2 to simulate execution of the plan from each initial state. If a safety constraint is violated, the L2 inference machinery can return the subset of the plan that is causing the constraint to be unsatisfied. This subset is then inserted as a nogood into the propositional representation of the planning problem that Blackbox has generated. That is to say, Blackbox has cast the planning problem as a propositional satisfiability problem, and we have added constraints to the problem that rule out a class of solutions as unsafe. We then re-invoke Blackbox’s satisfiability engine to find a new plan that satisfies the original planning problem and does not include any plan fragments that are known to be nogoods. The process of considering a solution and using the resulting nogoods to limit the space of feasible solutions is analogous to the conflict-directed search methods that provide a massive speed up in conflict-based diagnosis. Whether or not the proposed technique rapidly focuses the safe planning problem on a feasible solution in the same manner will to a great extent depend upon the nature of the search space and the generality of the conflicts found. That is to say, if a conflict only rules out the single plan that generated it, there is no speed up. If a conflict rules out an exponential number of similar solutions from consideration, as it can in diagnosis, then the speed up can be immense. We have the infrastructure in place to perform conflict-based repair of plans generated by Blackbox and have run simple experiments. The task of investigating what types of conflicts the L2 engine should return in order to quickly focus the planning process remains.

If empirical results suggest that the conflicts we are able to generate are not sufficiently focusing the satisfiability problem to produce a repaired plan, we will investigate techniques for safe generation. In safe generation, we must consider all possible current states when determining the safety of an action, in effect reasoning across multiple possible worlds. This is similar in spirit to techniques from conformant planning, wherein one must reason across multiple worlds in order to ensure all possible initial states reach the goal. The multiple-world graphplan structures used in Conformant Graphplan (Smith & Weld 1998) and the binary decision tree representations of the set of possible executions used in conformant planning

via model checking (Cimatti & Roveri 1999) are two possible optimizations of the safe planning process inspired by the conformant planning literature. In this case, we will leverage the special features of our domain (e.g., the fact that we are simply preserving safety rather than goal achievement across the multiple worlds) in order to scale these techniques to the problem sizes we face.

Chapter 9

Potential Areas for Future Work

In the preceding chapters, we introduced a problem formulation and representation for both discrete state identification and safe planning. We provided algorithms for addressing the state identification problem, and detailed the results of running those algorithms on realistic problems from the spacecraft domain. Those algorithms performed well in certain cases, but it is unsound and evidence suggests there is a well-defined subset of the problem space that causes a serious degradation in performance. The insights from these tests motivate a number of algorithmic improvements described below. In addition, we have not yet provided any algorithms for the safe planning problem or more general problems that violate the assumptions of the safe planning formulation. These potential areas for future research are described below.

9.1 Safe Planning

We described three variations of the safe planning problem. Each of these extend STRIPS-style planning by allowing a disjunction over the starting conditions, a set of safety constraints that need not hold in every state but must be monotonically increasing, and action costs. Recently, compilation of a STRIPS-style planning problem to a graphical representation (*e.g.*, Graphplan **cite**) or directly into a propositional satisfiabilty problem (*e.g.*, SATPLAN **cite**) has been shown to be a viable approach to planning. Conformant Graphlan **cite** solves the safe planning problem with unit costs, no safety constraints and the restriction that all start states reach the goal state. We propose to address the safe planning problem by generalizing conformant graphplan to handle these additional problem features, or by similarly extending SATPLAN-style approaches.

9.2 Adding Soundness to our Conservative State Identification Approximation

The technique of selective model extension, wherein we represent only those state variables whose failure could be witnessed by a command, is complete but not sound. In using this technique to vastly cut down the size of the representation and the search space, we eliminate past observations that would otherwise rule out imposter trajectories. One solution is to simulate each hypothesized trajectory against the observations, thus ruling out any inconsistent trajectories. Conceptually, a simulation of a trajectory is simply a consistency check of the $\mathcal{T}$ assigment and is linear in the length of the model. Performing a complete but unsound search followed by a rapid soundness check therefore seems a promising approach.

However, our optimized trajectory representation depends upon the fact that without observations, in many cases a failure at time t is equivalent to the same failure at times $t-1$ or $t+1$. That is, every failure represents a window during which time the failure may have occurred. When attempting to check a trajectory for consistency against the observations that actually occurred, we cannot rule out a trajectory until we have checked it against every arrangment of its failures to times within the time windows the

failures represent. While simulating a trajectory is a linear process, it is an open question as to whether we can avoid checking an exponential number of time-varying variations of the hypothesized trajectories.

9.3 Interleaving Coverage Generation and Consistency Checking

We have seen that adding conflicts to the conflict coverage problem can radically reduce the amount of time it takes for the *GenerateCover* algorithm to return a set of covers and for those covers to be checked for consistency. In fact, if the conflicts discovered when checking the covers generated by a call to *GenerateCover* are added to the *GenerateCover* call, *GenerateCover* returns just the consistent covers almost immediately. Interleaving consistency checking and conflict coverage may significantly cut down on the number of candidates returned by *GenerateCover* and the amount of time both generation and consistency checking require.

The *GenerateCover* algorithm is currently implemented as a recursive procedure, building full covers from the results of a smaller coverage problem. Thus no full covers are available until all covers are returned by the top level *GenerateCover* invocation. Recasting *GenerateCover* as an iterative algorithm will allow it to generate a number of full covers then check them for consistency, then add the conflicts to the conflicts being covered and continue. Note that this does not effect the final output of the algorithm. In the recursive formulation, all covers are checked for consistency in the final step, and any that contain a conflict are eliminated. In the iterative formulation, conflicts are discovered earlier, and covers that contain known conflicts are not generated. Some experimentation will be required to discover the appropriate interleaving of cover generation and conflict checking.

9.4 Finite Horizons

A simple fixed time horizon limits the size of the transition system, and thus the amount of search that can be done. As a side effect, it may also cut off consideration of overlapping failures in the past, as seems to cause the largest performance degradations. The algorithms and implementations for a simple time horizon are in place but have not yet been tested and investigated. A more interesting approach is to choose the horizon more intelligently. Iteratively deepening the horizon would support adjusting the search as time or uncertainty requires. Only exploring each device's history up to the last point it was considered to have failed should reduce the explosion in possible trajectories. Both of these approaches require introducing a recency bias into the probability assignments to transition choices in order to be well founded.

9.5 Reintroducing Observations

We have recently augmented the *L2* implementation to allow $\mathcal{M}_\Sigma$ and $\mathcal{O}$ to be duplicated at each time step. This of course causes the transition system size to grow at an unmanageable rate. One alternative is to duplicate $\mathcal{M}_\Sigma$ and $\mathcal{O}$ at a small number of recent time steps to increase the likelihood that imposter trajectories will be considered. This approach is largely implemented and will be tested and investigate shortly. A more interesting approach is to selectively introduce portions of $\mathcal{M}_\Sigma$ and $\mathcal{O}$ in the same manner that the transition system selectively introduces only those portions of $\mathcal{M}_\mathcal{T}$ and Σ that are relevant at each step. In the case of $\mathcal{M}_\mathcal{T}$ this is rather simple. One or a few command assignments are made at each time step, rendering most of $\mathcal{M}_\mathcal{T}$ irrelevant. Typically all observations are available at every time step, making the question of what portion of $\mathcal{M}_\Sigma$ to eliminate somewhat more complex. Ideally we would like to introduce only those portions of $\mathcal{M}_\Sigma$ and $\mathcal{O}$ that will enable us to distinguish between two possible trajectories. An approach then is to introduce the portions of $\mathcal{M}_\mathcal{T}$ and Σ that are relevant given the commanding history, then introduce only those portions of $\mathcal{M}_\Sigma$ and $\mathcal{O}$ that distinguish between possible values of the introduced subset of Σ. Techniques such as prime implicant generated used to reduce the size of the introduced $\mathcal{M}_\mathcal{T}$ could also be used to reduce $\mathcal{M}_\Sigma$. Alternatively, if the introduced $\mathcal{M}_\Sigma$ is yet

too large, we will investigate characterizing how the size of the introduced portion of $\mathcal{M}_\Sigma$ can be traded against diagnosis speed and specificity when searching for trajectories.

9.6 Active Testing

Model-based diagnosis gives us a lot of work on probe selection. In addition to the usual state determination techniques of MBD, we can issue commands. Thus we should look at distinguishing sequences as well. That is, in the current possible states there may be no sensing action we can perform that disambiguates the states. However, if we issue command C, the result from each of our possible states may be a new state such that the new set has a distinguishing sensing action.

9.7 Uniform Distributions

We have the equivalent of the "kidnapped robot" problem. In the kidnapped robot problem, the robot has to navigate starting with completely uniform belief state. It may be in any location with equal probability. The equivalent problem is a reboot of a spacecraft processor in flight, erasing the knowledge the system accumulated about the state of the device. When we encounter a reboot or the kidnapped robot problem, we will have a uniform distribution over all states. Since we are only representing the few most likely states, we cannot represent this distribution. We will use the compositionality of the system to force devices into a known state in the appropriate order. Commanding devices are controlled before the corresponding commanded device. We will use the model to generate a homing sequence for each device, given that the upstream devices are in known modes.

Chapter 10

Summary

10.1 Results Thus Far

The following items have been achieved by the author thus far

- Formulation of the trajectory tracking problem
- Correspondence to POMDP
- Transition system representation
- Approximate representations
- Conflict coverage algorithm
- Algorithm implementation
- Experimental results in trajectory tracking
- Formulation of the safe planning problem

10.2 Proposed Work

10.2.1 Action Selection

- Mapping of transition system to Graphlan or SATPLAN
- Algorithm for maximally safe planning
- Algorithm for single goal-safe planning
- Algorithm implementation
- Experimental results in safe planning

10.2.2 State Identification

- Finite horizon experiments with state identification
- Improved state identification through complete soundness check or re-introduction of specific variables

We are currently investigating these and other extensions to *L2*. The resulting system will be evaluated on Earth-bound testbeds representing an interferometer and a Mars propellant plant. In addition, it will be flown as an experiment on the X-34 rocket plane in 2001 and the X-37 orbital vehicle in 2002.

References

Bernard, D. E.; Dorais, G. A.; Fry, C.; Jr., E. B. G.; Kanefsky, B.; Kurien, J.; Millar, W.; Muscettola, N.; Nayak, P. P.; Pell, B.; Rajan, K.; Rouquette, N.; Smith, B.; and Williams, B. C. 1998. Design of the remote agent experiment for spacecraft autonomy. In *Procs. IEEE Aerospace*.

Boyen, X., and Koller, D. 1998. Tractable inference for complex stochastic processes. In *Procs. UAI-98*, 33–42.

Cassandra, A.; Kaelbling, L. P.; and Kurien, J. 1996. Discrete bayesian uncertainty models for mobile-robot navigation. In *IROS96*.

Cheng, H.-T. 1988. *Algorithms for Partially Observable Markov Processes*. Ph.D. Dissertation, University of British Columbia.

Cimatti, A., and Roveri, M. 1999. Conformant planning via symbolic model checking. In *Proceedings of the European Conference on Planning (ECP99)*.

de Kleer, J., and Williams, B. C. 1987. Diagnosing multiple faults. *Artificial Intelligence* 32(1):97–130. Reprinted in (Hamscher, Console, & de Kleer 1992).

de Kleer, J., and Williams, B. C. 1989. Diagnosis with behavioral modes. In *Proceedings of IJCAI-89*, 1324–1330. Reprinted in (Hamscher, Console, & de Kleer 1992).

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269–271.

Dressler, O., and Struss, P. 1992. Back to defaults: Characterizing and computing diagnoses as coherent assumption sets. In *Procs. ECAI-92*.

Friedman, N., and Halpern, J. Y. 1999. Modeling belief in dynamic systems part ii: Revision and update. *JAIR* 10.

Goldszmidt, M., and Pearl, J. 1992. Rank-based systems: A simple approach to belief revision, belief update, and reasoning about evidence and actions. In *Procs. KR-92*, 661–672.

Hamscher, W.; Console, L.; and de Kleer, J. 1992. *Readings in Model-Based Diagnosis*. San Mateo, CA: Morgan Kaufmann.

Kautz, H., and Selman, B. 1999. Unifying sat-based and graph-based planning. In *Proceedings of IJCAI-99*.

Kurien, J., and Nayak, P. P. 2000. Back to the future with consistency based trajector tracking. In *Proceedings of AAAI-00*.

Kurien, J.; Nayak, P. P.; and Williams, B. 1998. Model-based autonomy for robust mars operations. In *Founding Convention of the Mars Society*.

Littman, M.; Cassandra, A.; and Kaelbling, L. 1995. Learning policies for partially observable environments: Scaling up. In *Machine Learning: Proceedings of the Twelfth International Confernece*, 362–370.

Muscettola, N.; Nayak, P. P.; Pell, B.; and Williams, B. C. 1998. Remote Agent: To boldly go where no AI system has gone before. *Artificial Intelligence* 103:5–47.

Nayak, P. P., and Williams, B. C. 1997. Fast context switching in real-time propositional reasoning. In *Proceedings of AAAI-97*.

Nourbakhsh, I.; Powers, R.; and Birchfield, S. 1995. Dervish: An office-navigating robot. *AI Magazine* Summer.

Pell, B.; Gamble, E. B.; Gat, E.; Keesing, R.; Kurien, J.; Millar, B.; Nayak, P. P.; Plaunt, C.; and Williams, B. C. 1998. A hybrid procedural/deductive executive for autonomous spacecraft. In *Proceedings of the Second International Conference on Autonomous Agents*.

Smith, D. E., and Weld, D. S. 1998. Conformant graphplan. In *Proceedings of AAAI-98*.

Sondik, E. J. 1971. *The Optimal Control of Partially Observable Markov Processes*. Ph.D. Dissertation, Stanford University.

Struss, P. 1997. Fundamentals of model-based diagnosis of dynamic systems. In *Procs. IJCAI-97*, 480–485.

Williams, B. C., and Nayak, P. P. 1996. A model-based approach to reactive self-configuring systems. In *Procs. AAAI-96*, 971–978.

Williams, B. C., and Nayak, P. P. 1997. A reactive planner for a model-based executive. In *Procs. IJCAI-97*.